Sistemas Informáticos

Curso 2010-2011

Optimización térmica del banco de registros mediante técnicas de computación evolutiva y simulación de eventos discretos

Alumnas:
Esperanza Hidalgo López
Maribel Lértora Ginés

Director:
José Luis Risco Martín

Facultad de Informática

Universidad Complutense de Madrid

Autorización

Autorizamos a la Universidad Complutense de Madrid a utilizar y/o difundir con fines académicos y no comerciales, siempre mencionando expresamente a sus autores, tanto la propia memoria como el código, la documentación y/o el prototipo desarrollado.

Madrid, a 1 de Julio de 2011

Esperanza Hidalgo López

Maribel Lértora Ginés

Agradecimientos

En primer lugar queremos agradecer a nuestras familias por su constante motivación, sobre todo, porque gracias a ellos hemos conseguido llegar a ser quienes somos.

No nos podemos olvidar de cada uno de los compañeros que nos han acompañado a lo largo de esta etapa, junto a ellos hemos vivido momentos que perdurarán siempre en nuestro recuerdo.

Por último queremos hacer una mención especial a nuestro director José Luis, por su continua paciencia y por brindarnos todo tipo de apoyo, conocimiento y ayuda a lo largo del proyecto ("porque 20 años programando se tienen que notar"), haciéndolo lo más llevadero posible.

A todos ellos, gracias.

Palabras clave

Algoritmo genético, DEVS, estudio térmico, MIPS, MOEA.

Resumen

El procesador MIPS es utilizado en cursos de arquitectura de computadores para explicar materias tales como análisis de rendimiento, consumo de energía o fiabilidad.

Por otro lado, el formalismo DEVS es un acrónimo del inglés para referirse a Discrete Event System Specification, o lo que es lo mismo Especificación de Sistemas de Eventos Discretos. El término es ahora estándar en el campo de la simulación para referirse a un formalismo modular y jerárquico, muy utilizado para modelar y analizar sistemas de diversos tipos.

En la actualidad, ya que cada vez se desean obtener computadoras más potentes, es interesante conocer de qué manera se pueden redistribuir ciertos componentes de modo que el calor desprendido no sea excesivo y tal que el coste de enfriamiento no sea muy alto.

En este proyecto se realiza un estudio térmico del modelo MIPS utilizando DEVS. Concretamente, por facilidad a la hora de obtener conclusiones y dado que se trata de un proyecto relativamente pequeño, desarrollamos una metodología que, mediante políticas de reasignación de registros basadas en algoritmos genéticos, disminuya considerablemente la temperatura final del Banco de Registros.

Abstract

The MIPS processor is used in computer architecture courses in order to explain matters such as performance analysis, energy consumption or reliability.

On the other hand, DEVS is a formalism used for modeling and analyzing discrete event systems. This is a standard term in the field of simulation referring to a modular and hierarchical formalism that is very used to model and analyze many different systems.

Nowadays, due to the desire for more powerful computers, it is interesting to learn how to reallocate certain components in order to achieve heat reduction, with low cooling costs.

In this project we carry out a thermal analysis of the MIPS processor using DEVS. Specifically, because this is simpler if we intend to draw conclusions and because this is a relatively small project, we develop a methodology which, through register reallocation policy based on genetic algorithms, notably decreases the resulting Register Bank temperature.

Índice

1. Introducción y trabajo relacionado

1.1 Introducción

La tecnología evoluciona y crece a un ritmo exponencial en los últimos tiempos, lo mismo ocurre en el campo de la computación donde cada vez se desean máquinas que trabajen a un ritmo cada vez superior al anterior y cuya capacidad de almacenamiento vaya en aumento.

La plena informatización de la información ha sido la causa de que se deseen crear bases de datos muy potentes que necesitan de servidores capaces de ofrecer el servicio al ritmo que se pide.

A este problema le unimos al incremento de la capacidad de computación, y que por otro lado se desea que la tecnología se desarrolle en un espacio cada vez más pequeño. Nos es imposible concebir, por ejemplo, un ordenador personal de 4 núcleos de procesamiento cuyo tamaño sea dos veces un ordenador de un sólo procesador.

Si se desean conseguir escalas de integración cada vez más pequeñas se produce, entre otras cosas, un aumento del consumo de potencia. Cada uno de estos componentes anteriormente nombrados consume una potencia que, unido al consumo de potencia de los componentes que tiene a su alrededor, hace que no se rentable para realizar la tarea para la que había sido creado.

En nuestro proyecto queremos paliar estos efectos diseñando algoritmos de reasignación de registros. Para ello nos hemos centrado en un componente de la ruta de datos del procesador MIPS: el banco de registros. Es un buen ejemplo ya que, a pequeña escala representa a la perfección el problema explicado anteriormente.

Tenemos un conjunto de 32 registros en un espacio pequeño como es el banco de registros, y se desea encontrar la mejor manera de redistribuirlos para que la potencia consumida sea mínima.

El proceso que hemos seguido en este estudio ha sido el siguiente, se repite para los tres ejemplos utilizados:

1. Ejecutar el algoritmo en su modo general y obtener la temperatura de cada registro, el valor máximo, mínimo y medio.

2. Cambiar la configuración del Banco de registros, y repetir el paso 1 para cada uno de los diferentes modelos, correspondientes a las distintas configuraciones. Se observa que hay organizaciones que disminuyen la temperatura máxima (es en la que nos fijaremos) mientras que otras sólo lo empeoran.
   Esto es debido a que con algunas configuraciones *acercan* entre sí registros muy utilizados en la ejecución y, por lo tanto, el calor de ambos perjudica a todos los demás.

3. Con cada uno de los modelos del apartado anterior, aplicamos un algoritmo genético cuyo objetivo sea el de redistribuir los registros por el banco de registros para que la temperatura máxima sea mínima. En ese paso evitaremos lo que en ocasiones obteníamos en el paso 2, esto es que dos registros muy calientes permanecieran juntos.

Los algoritmos genéticos se inspiran en la evolución biológica y su base genético-molecular. Estos algoritmos hacen evolucionar una población de individuos sometiéndola a acciones aleatorias semejantes a las que actúan en la evolución biológica (mutaciones y recombinaciones genéticas), así como también a una selección de acuerdo con algún criterio, en función del cual se decide cuáles son los individuos más adaptados, que sobreviven, y cuáles los menos aptos, que son descartados.

En las siguientes imágenes se explican los pasos aplicados en nuestro proyecto. En la Figura 1 se ve cómo los registros de color azul son aquellos que han sido poco utilizados por el algoritmo y, por lo tanto, su temperatura es fría. Sin embargo aquellos en color rojo están muy calientes debido a su uso prolongado por parte de la ejecución:

R0	R16
R1	R17
R2	R18
R3	R19
R4	R20
R5	R21
R6	R22
R7	R23
R8	R24
R9	R25
R10	R26
R11	R27
R12	R28
R13	R29
R14	R30
R15	R31

Figura 1 Banco de registros tras ejecutar un algoritmo

R0	R8	R16	R24
R1	R9	R17	R25
R2	R10	R18	R26
R3	R11	R19	R27
R4	R12	R20	R28
R5	R13	R21	R29
R6	R14	R22	R30
R7	R15	R23	R31

Figura 2 Distribución 8x4

R22	R8	R16	R24
R1	R9	R17	R19
R15	R10	R18	R26
R3	R30	R20	R27
R4	R12	R25	R13
R5	R28	R21	R29
R6	R14	R0	R11
R7	R2	R23	R31

Figura 3 Reasignación de los registros

Los resultados obtenidos en este estudio pueden ser usados posteriormente por el compilador de manera que, por ejemplo, donde antes se utilizaba el registro R0 ahora se utilizará el registro R22.

Tras realizar estos ejemplos, se llega a la conclusión que podemos reducir la potencia consumida gracias a una buena organización de los componentes en vez de invertir grandes sumas de dinero en el enfriamiento. Esto es gracias, en gran parte, al uso de la simulación para realizar estos estudios.

Si comparamos la simulación con experimentos reales usando hardware, la simulación posee muchas más ventajas (facilidad de uso, de configuración y de modificación) además de tener un coste mucho menor.

Para realizar la simulación se ha utilizado el modelo del MIPS en su versión monociclo, y ha sido desarrollado usando DEVS. Se ha escogido como formalismo DEVS porque aparte de cumplir todos los requisitos relevantes en cuanto a técnicas de simulación y modelado, es muy apropiado para modelar sistemas de arquitecturas de computadores.

1.2 Trabajo relacionado

Para ejecutar los ejemplos de estudio en los que centraremos este proyecto, se ha necesitado de un modelo MIPS monociclo.

Antes de comenzar a implementarlo desde el principio, observamos que unos compañeros de esta misma universidad hicieron este mismo trabajo para la asignatura de Sistemas Informáticos del curso pasado (Calvo Valdés, 2010).

El objetivo de su proyecto era implementar el procesador MIPS usando DEVS en sus versiones monociclo, multiciclo y segmentado para facilitar la tarea docente. En la Figura 4 se muestra el entorno desarrollado por nuestros compañeros.

En este caso sólo necesitamos la primera versión, ya que el objetivo de este proyecto no es la implementación del procesador MIPS, porque es su versión más sencilla de entender y crear. Por este motivo, hemos partido de su procesador para modificarlo según nuestras necesidades.

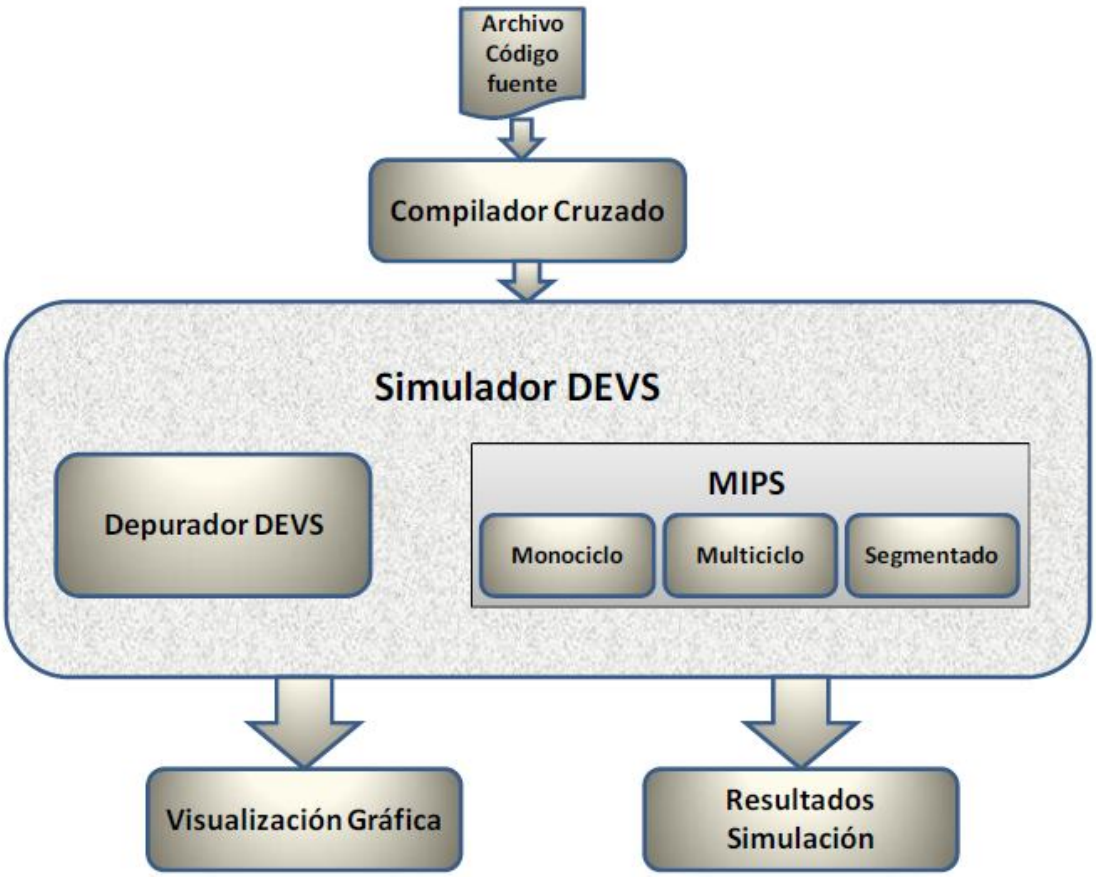

Figura 4 Entorno de simulación desarrollado Proyecto 09/10

El tema principal de estudio de este proyecto es el estudio térmico aplicado al banco de registros del procesador MIPS. Sobre tema hay infinidad de estudios y artículos. Para nuestro proyecto hemos partido de un artículo (Brooks, 2007) en el que se explica que un buen modelado previo reduce el impacto térmico.

Este artículo se basa en las relaciones entre el consumo de potencia, la temperatura, la fiabilidad y la ejecución del proceso.

A continuación se muestra dicha relación en la Figura 5, pero de manera más gráfica. Las cajas verdes representan atributos que no influyen en los demás, como por ejemplo el coste de la electricidad. Por otro lado las cajas azules indican qué procesos tienen algún atributo que puede influir en otros.

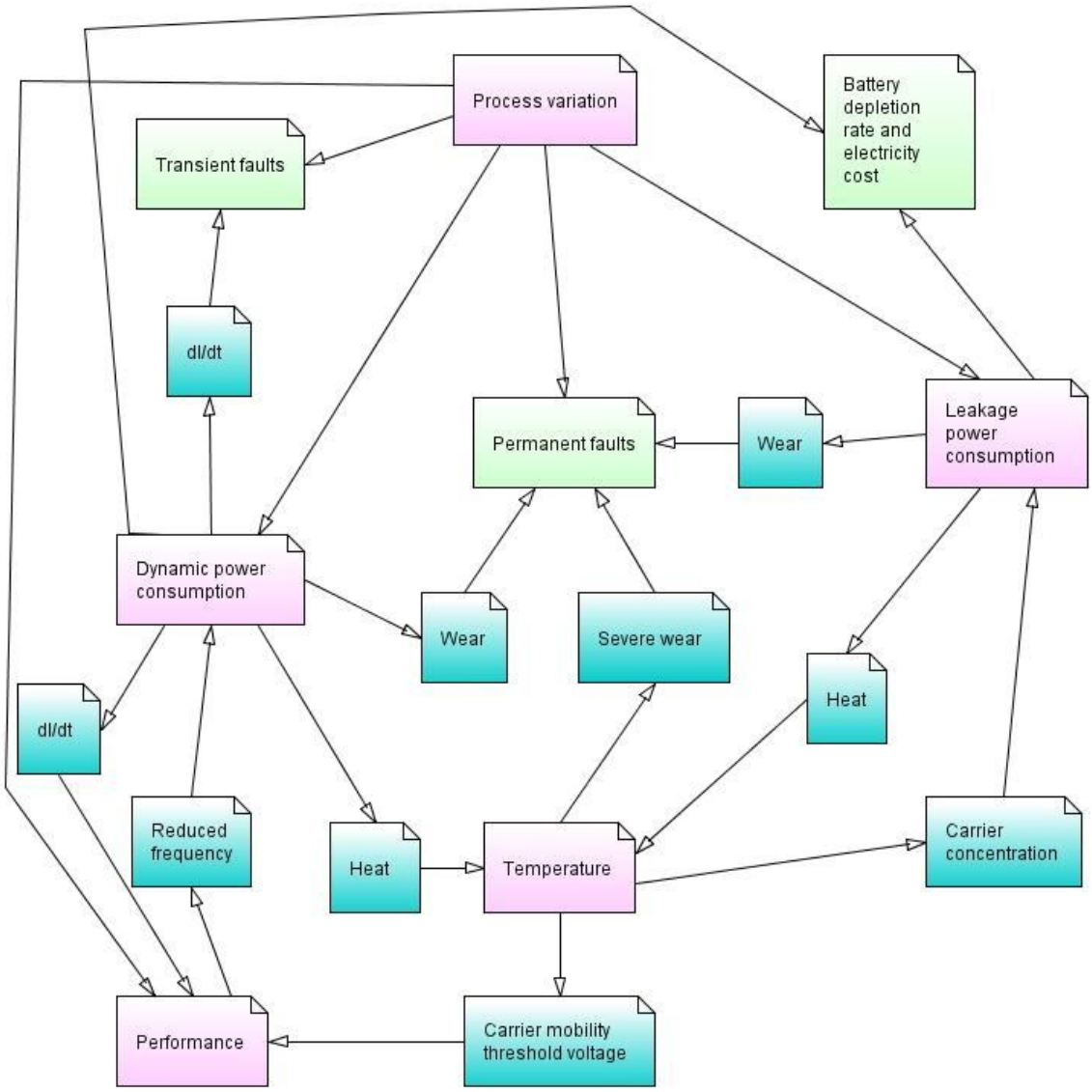

Figura 5 Brooks 2007

En nuestro caso la solución también reside en el modelado ya que reasignamos los registros, aunque nuestro trabajo está centrado sólo en el problema de la temperatura.

2. Contribuciones del proyecto

El estudio térmico no es la única contribución de este trabajo, sino el paso final de un largo proceso.

Desarrollamos un método para obtener una política de reasignación de registros, de forma que la temperatura final del banco de registros se minimice considerablemente. Hemos usado como arquitectura objetivo el procesador MIPS32, aunque el estudio es extensible a cualquiera.

El estudio se divide en varias fases, explicadas a continuación.

En primer lugar obtenemos el patrón de accesos a los registros mediante simulación. Hemos desarrollado un simulador del procesador usando DEVS, que es un estándar en el campo de la simulación muy utilizado para analizar y modelar sistemas.

Se crea una clase que extienda al MIPS, aunque en nuestro caso únicamente nos centramos en las conexiones en las que interviene el banco de registros. Para ello haremos uso de un transductor, de manera que las entradas y salidas de este también sean las del banco de registros.

En segundo lugar calculamos el impacto energético de ese patrón de accesos. Desde el propio simulador se obtiene los valores de la energía y la potencia de cada registro, necesarios para los pasos posteriores.

Con el resultado anterior ejecutamos un algoritmo de optimización, a cada uno de los benchmarks, que obtiene una política de reasignación de registros.

Está implementado como un algoritmo genético multi-objetivo (MOEA), cuyo objetivo será el de separar lo más posible los registros cálidos entre sí, de esa manera se verá reducida la temperatura del banco de registros. Cada uno de los alelos será un registro y cada cromosoma será una configuración concreta del banco de registros.

Finalmente validamos la política de reasignación mediante simulación térmica. Con este fin hemos implementado un modelo térmico previamente publicado en (Brooks, 2007).

Cada uno de estos pasos, se explicarán más detalladamente en los posteriores apartados. La siguiente imagen Figura 6 muestra un resumen esquematizado de las contribuciones de este proyecto

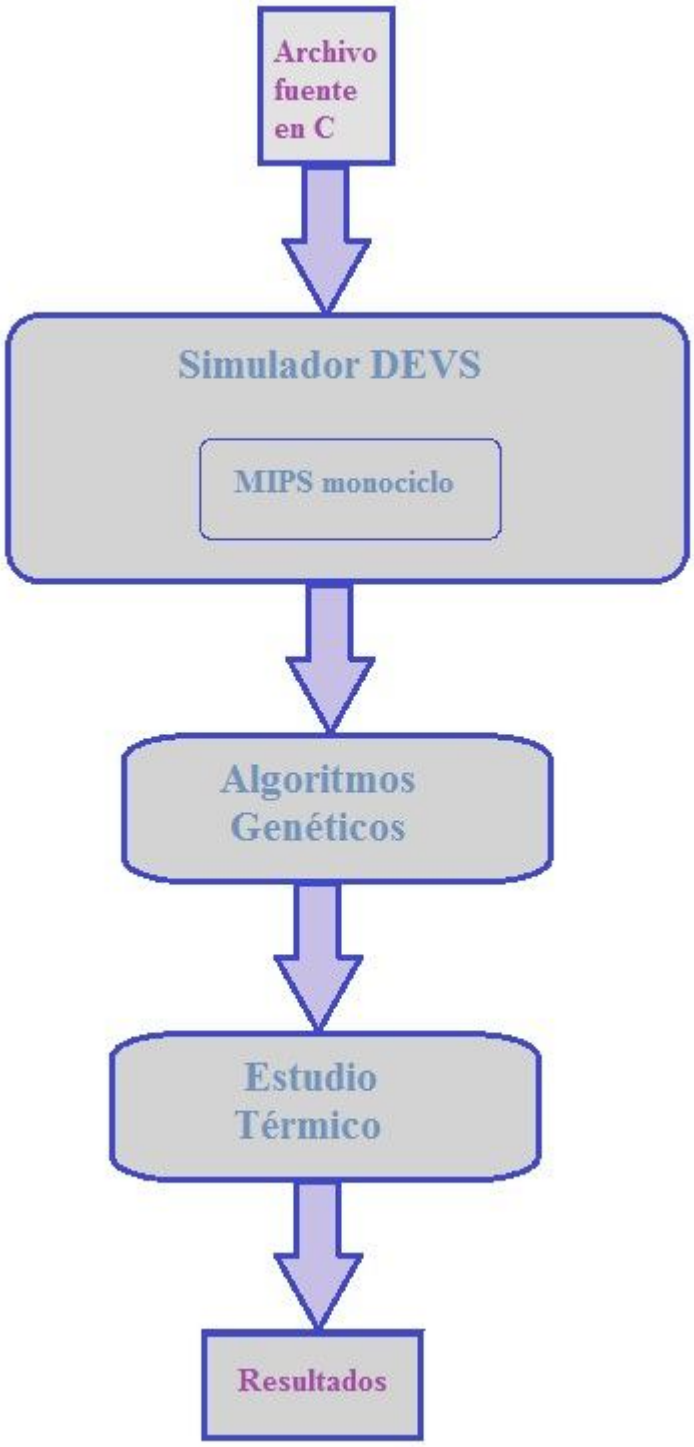

Figura 6 Contribuciones del proyecto

3. Definición del procesador MIPS32

3.1. Introducción

Como se ha comentado anteriormente, el propósito de este proyecto es el estudio térmico del procesador MIPS, aunque por comodidad se haya centrado el trabajo en el Banco de registros, para mejorar la distribución de sus componentes y evitar un calentamiento excesivo. Es por ello que no se ha implementado todo el repertorio de instrucciones, sino que nos hemos centrado en aquellas más relevantes de la unidad entera.

Uno de los libros más importantes, y a su vez más utilizado para la docencia, sobre el procesador MIPS es el (Patterson, 2007), aunque éste varía ligeramente con respecto a las diferentes ediciones del libro.

Se utilizará a continuación, como base para este proyecto, la ruta de datos del MIPS32 que se explica en varias asignaturas de arquitectura de computadores.

3.2. Descripción general

Con el nombre de MIPS, que son las siglas de *Microprocessor without Interlocked Pipeline Stages*, se conoce a toda una familia de microprocesadores de arquitectura RISC desarrollados por MIPS Technologies.

Es un tipo de microprocesador con las siguientes características fundamentales:

1) Instrucciones de tamaños fijos y presentados en un reducido número de formatos.
2) Sólo las instrucciones de carga y almacenamiento acceden a la memoria por datos.

Las primeras arquitecturas MIPS fueron implementadas en 32 bits (generalmente rutas de datos y registros de 32 bits de ancho), si bien versiones posteriores fueron implementadas en 64 bits.

Debido a que los diseñadores crearon un conjunto de instrucciones tan claro, los cursos sobre arquitectura de computadores en universidades y escuelas técnicas a menudo se basan en la arquitectura MIPS. El diseño de la familia de CPU's MIPS influiría de manera importante en otras arquitecturas RISC posteriores como los DEC Alpha.

En el año 1981, John L. Hennessy y su equipo en la Universidad de Stanford comenzaron a crear el primer procesador MIPS. La idea que les impulsó a ello era mejorar de manera drástica el rendimiento del procesador haciendo uso de la segmentación. En aquellos años se trababa de una técnica conocida, aunque complicada a la hora de implementar.

Estos procesadores suelen disponer de muchos registros de propósito general para ser empleados en la ejecución, minimizando de esta manera los accesos a memoria cada vez que se necesita un dato.

El objetivo de diseñar máquinas con esta arquitectura es posibilitar la ejecución y el paralelismo en la ejecución de instrucciones y reducir los accesos a memoria para aumentar el rendimiento y la velocidad de ejecución. Las máquinas RISC protagonizan la tendencia actual de construcción de microprocesadores. PowerPC, DEC y Alpha son ejemplos de algunos de ellos.

La técnica de la segmentación consiste en dividir la ejecución de una instrucción en varias etapas. A diferencia de los procesadores con un diseño tradicional, los cuales esperaban a que una instrucción finalizase antes de lanzar la siguiente, en los procesadores segmentados se permite lanzar la primera etapa de una instrucción cuando aún se está ejecutando la anterior. Con los diseños antiguos ocurría a menudo que grandes áreas de la CPU permanecían inactivas mientras el proceso continuaba. Por otra parte, la frecuencia de reloj venía determinada por la latencia de ciclo completo, mientras que con el diseño segmentado viene determinada por el camino crítico, es decir, por la etapa de segmentación que tarda más en completarse.

Por otra parte, debido a la segmentación, se vio en la necesidad de introducir bloqueos para que las instrucciones que necesitan varios ciclos de reloj para completarse dejen de cargar datos desde los registros de segmentación. Estos bloqueos pueden durar mucho tiempo y no permitirían posteriores mejoras en la velocidad. Uno de los objetivos que se marcaron los creadores del MIPS fue el de que todas las etapas de todas las instrucciones tardasen un único ciclo en completarse. Con este cambio, operaciones como la multiplicación y la división resultaban más largas y se desechaban algunas instrucciones útiles, pero en conjunto el rendimiento general se vería mejorado.

Como consecuencia de la eliminación de instrucciones, se creó una polémica y muchos especialistas pronosticaron que este proyecto jamás alcanzaría los objetivos propuestos.

En la actualidad estos procesadores se emplean a menudo para la fabricación de sistemas empotrados, en dispositivos para Windows CE, en routers CISCO, en videoconsolas, etc. Ha pasado por muchas revisiones que han aumentado su rendimiento (tamaño de memoria, velocidad de reloj…) desde el MIPS y el MIPS V hasta las actuales MIPS32 y MIPS64, que trabajan con 32 y 64 bits respectivamente. Incluso se ha logrado implementar una versión de MIPS súper-escalar (R800 en 1994) que es capaz de ejecutar 2 instrucciones de memoria y de la ALU en un solo ciclo.

Sin embargo, el uso del MIPS como procesador principal de estaciones de trabajo ha caído, aunque por otra parte el uso de microprocesadores MIPS en sistemas empotrados es probable que se mantenga gracias al bajo consumo de energía y características térmicas de las implementaciones integradas, así como a la gran disponibilidad de herramientas de desarrollo y de expertos conocedores de la arquitectura.

3.3. Funcionamiento

Para el desarrollo de la memoria, se va a analizar a fondo el funcionamiento del procesador monociclo, esto es un ciclo por instrucción y tiempo de ciclo largo, incorporando en el proceso las instrucciones implementadas en el simulador.

3.4. Formato de instrucción máquina

Todas las instrucciones del repertorio del MIPS tienen 32 bits de anchura y son de 3 tipos:

1) Instrucciones aritméticas (formato R), que como su propio nombre indica se encargan de realizar operaciones aritméticas.
2) Instrucciones con referencia a memoria (formato I), que sirven para almacenar datos en memoria o extraer datos de la misma.
3) Instrucciones de salto (formato J), que sirven para realizar bifurcaciones en el programa.

En la Figura 7 se representan los formatos de las 3 instrucciones anteriormente descritas, indicando cuántos bits corresponden a cada campo. Los campos de la instrucción son por tanto:

1) **op:** identificador de instrucción
4) **rs,rt,rd:** identificadores registros fuente/destino
5) **desp:** cantidad a desplazar (en operaciones de desplazamiento)
6) **funct:** selecciona la operación aritmética a realizar
7) **dirección:** dirección del salto

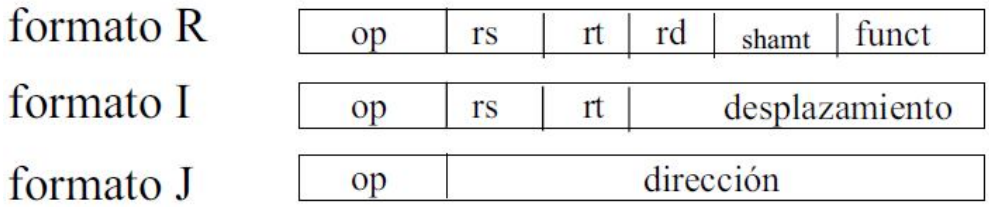

Figura 7 Formatos de instrucción

3.5. Procesador monociclo

Todas las instrucciones tardan un ciclo de reloj (CPI=1) por lo tanto, para que esto pueda darse, el tiempo de ciclo debe ser igual o mayor que la instrucción más compleja. Esto hace que el ciclo de ejecución sea largo, ya que de lo contrario se obtendrían resultados inesperados debido a errores en los módulos internos.

Esto hace que la ejecución sea muy lenta, por lo que instrucciones muy complejas como la división y la multiplicación u operaciones con modos complejos de direccionamiento no pueden implementarse.

Por otro lado, cada unidad funcional sólo puede ser utilizada una vez por ciclo, lo que obliga a adaptar el hardware mediante el uso de multiplexores para reutilizar los bloques o, por ejemplo que la memoria se divida en dos módulos: memoria de datos y de instrucciones, de lo contrario sería imposible completar las lecturas y escrituras necesarias en un solo ciclo. Esta solución aumenta el coste del hardware.

Se puede mejorar el diseño, existen dos opciones:

- Implementación multiciclo
  El número de ciclos por instrucción (CPI) es variable, las instrucciones sencillas se ejecutan en un número menor de ciclos.
- Implementación segmentada
  Se solapa la ejecución de varias instrucciones, aunque se encuentran en distintas fases de ejecución.

Temporizador Monociclo

La idea fundamental en que se sustenta la ruta de datos monociclo es la existencia de dos *barreras* secuenciales entre las cuales se ejecuta toda la carga de trabajo combinacional y que constituyen un ciclo (y por ende, una instrucción), como ilustra la siguiente Figura 8. Los únicos módulos secuenciales que aparecen en la ruta son el Registro que actúa de contador de programa (PC), el Banco de registros y la Memoria de datos.

En la ejecución típica de una instrucción todos los registros deberán cargarse simultáneamente aunque de un modo selectivo, los valores se van propagando por las redes combinacionales hasta estabilizarse en las entradas de los registros. Este proceso se repetirá indefinidamente. Todo ello está regulando en todo momento por las señales producidas por la Unidad de Control.

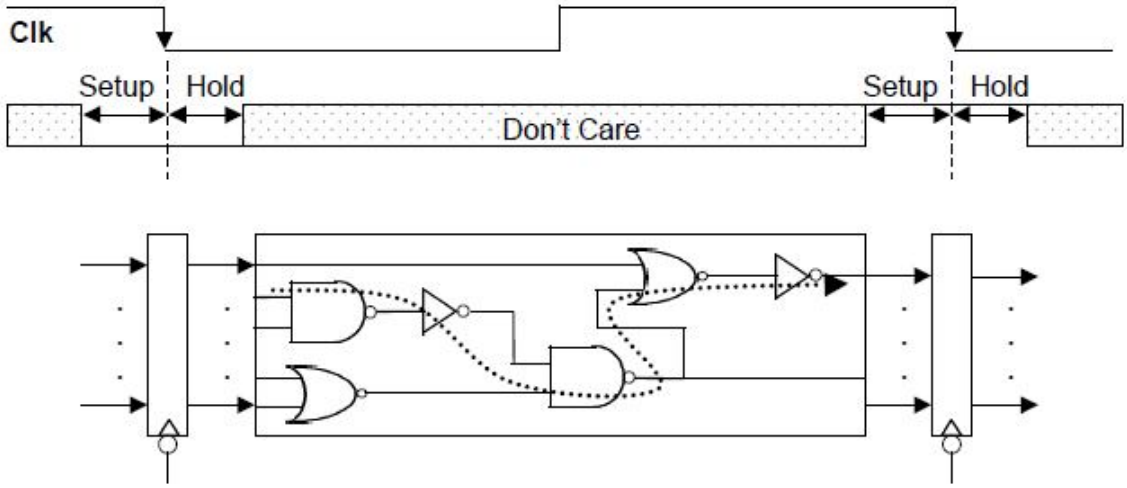

Figura 8 Temporización Monociclo

A continuación, se definen conceptos útiles para la temporización del procesador monociclo:

- **Setup:** Tiempo que debe transcurrir antes del flanco de reloj en el cual la entrada a un registro debe permanecer estable para evitar introducir valores incorrectos.
- **Hold:** Tiempo que debe transcurrir tras el flanco de reloj en el cual la entrada a un registro debe permanecer estable para evitar introducir valores incorrectos.
- **Clk-to-Q:** Es el tiempo que tarda en aparecer el valor correcto a la salida de un registro tras el flanco de reloj.
- **Clock Skew:** Es el retardo asociado a las señales de reloj de los distintos biestables, necesario para que funcionen de forma correcta a pesar de recibir el flanco de reloj con cierto retraso.
- **Tciclo:** Es el tiempo en el que se ejecuta una instrucción completa. Para permitir que se realicen todos los cálculos de forma correcta este debe ser el del camino más largo.

Todo lo anterior se resume en las siguientes dos ecuaciones:

- Tciclo = CLK-to-Q + (camino de máximo retardo) + setup + clock skew
- (CLK-to-Q + (camino de mínimo retardo – clock skew)) > Hold

Repertorio de instrucciones

En este apartado, se resumirán el repertorio de instrucciones MIPS utilizadas en nuestro proyecto.

addiu rs, rt, inmed

La instrucción *addiu* realiza la suma entre el contenido de un registro y un operando inmediato, el resultado es almacenado en otro registro.

9 (6)	rs (5)	rt (5)	Inmediato (15)

addu rd,rs,rt

La instrucción addu realiza la suma sin signo entre el contenido de dos registros (rs y rt) y almacena el resultado en rd.

0 (6)	rs (5)	rt (5)	rd (5)	Shamt (5)	0x21 (6)

beq rs, rt, desplazamiento

Si los registros rs y rt son iguales, se suma al contador de programa la cantidad indicada por el *desplazamiento.*

4 (6)	rs (5)	rt (5)	desplazamiento

bgez rs, desplazamiento

Salta si el registro rs es mayor o igual a cero. En ese caso, se añade al contador de programa el *desplazamiento.*

1 (6)	rs (5)	1 (5)	desplazamiento

blez rs, desplazamiento

Salta si el registro rs es menor o igual a cero. En ese caso, se añade al contador de programa el *desplazamiento.*

6 (6)	rs (5)	0 (5)	desplazamiento

bltz rs, desplazamiento

Se salta la cantidad indicada en *desplazamiento* si el registro *rs* es igual a cero.

1 (6)	rs (5)	0 (5)	desplazamiento

bne rs, rt, desplazamiento

Se comparan los registros rs y rt, si estos no son iguales se salta la cantidad indicada en *desplazamiento*, sino se pasa a la siguiente instrucción.

5 (6)	rs (5)	rt (5)	desplazamiento

j desplazamiento

Salto incondicional a la dirección indicada por *desplazamiento*.

2 (6)	desplazamiento (26)

jr rs

Salto incondicional a la instrucción almacenada en el registro *rs*.

0 (6)	rs (5)	0 (15)	8 (6)

lw rt, rs, desplazamiento

Busca en memoria la palabra almacenada en la dirección *rs* + *inmediato*, y la almacena en *rt*.

0x23 (6)	rs (5)	rt (5)	desplazamiento

movn rd, rs, rt

Mueve el registro *rs* al registro *rd* si el tercer registro, *rt* no es igual a cero.

0 (6)	rd (5)	rs (5)	rt (5)	0xb (11)

movz rd, rs, rt

Mueve el registro *rs* al registro *rd* si el tercer registro, *rt* es igual a cero.

0 (6)	rd (5)	rs (5)	rt (5)	0xa (11)

sll rd, rt, shamt

Desplazamiento lógico a la izquierda de *rt*, lo almacena en *rd*

0 (6)	rs (5)	rt (5)	rd (5)	shamt (5)	0 (6)

slt rd, rs, rt

Comprueba si *rs* es menor que *rt*, el resultado se almacena en *rd*.

0 (6)	rs (5)	rt (5)	rd (5)	0 (5)	0x2a (6)

slti rd, rs, inmediato

Comprueba si *rs* es menor que *inmediato*, el resultado se almacena en *rd*.

0xa (6)	rs (5)	rt (5)	inmediato (16)

sltiu rd, rs, inmediato

Comprueba si *rs* es menor que *inmediato*, siendo este sin signo. El resultado se almacena en *rd*.

0xa (6)	rs (5)	rt (5)	inmediato (16)

subu rd, rs, rt

Realiza la resta sin signo entre los registros *rs* y *rt*, el resultado de la misma se almacena en el registro *rd*.

0 (6)	rs (5)	rt (5)	rd (5)	0 (5)	0x23 (6)

sw rt, rs, desplazamiento

Toma la palabra que se encuentra en la dirección de memoria *rs* + *inmediato* y la almacena en el registro *rt*.

0x29 (6)	rs (5)	rt (5)	inmediato (16)

xor rd, rs, rt

Tomando como operandos los registros *rs* y *rt*, se realiza la operación de "O exclusiva". El resultado se guarda en *rd*.

beqz rsrc , etiqueta

Se trata de una pseudoinstrucción, en la cual salta se salta donde indique *etiqueta* si *rsrc* es igual a cero.

bnez rsrc , etiqueta

Se trata de una pseudoinstrucción, en la cual salta se salta donde indique *etiqueta* si *rsrc* es distinto de cero.

li rd, imm

Almacena el valor inmediato de *imm* en el registro *rd*. Es una pseudoinstrucción.

move rd, rs

Es una pseudoinstrucción que mueve el registro *rs* al registro *rd*.

Componentes de la ruta de datos

Para ejecutar las instrucciones anteriores, se necesitan los componentes definidos a continuación.

Contador de programa:

Al inicio de cada ciclo, almacena la dirección de memoria que contiene la instrucción que se va a ejecutar.

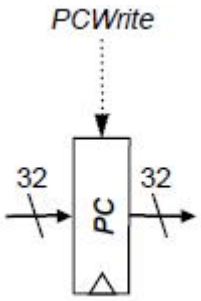

Figura 9 Contador de programa

Sumadores:

Se disponen de dos sumadores. Uno se encarga de sumar 4 al contador de programa, el otro suma el valor inmediato del salto al contador de programa.

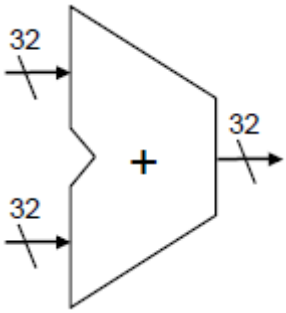

Figura 10 Sumador

ALU:

Capaz de realizar suma, resta, y-lógica, o-lógica, comparación de mayoría e indicación de que el resultado es cero (para realizar la comparación de igualdad mediante resta).

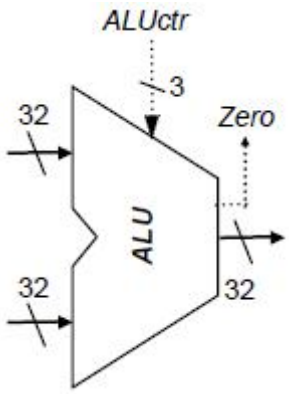

Figura 11 Unidad aritmético-lógica

Extensor de signo:

Para adaptar el operando inmediato de 16 bits al tamaño de palabra, de 32 bits.

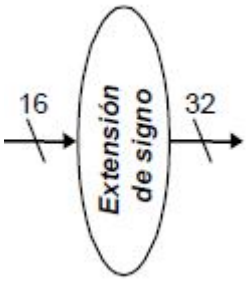

Figura 12 Extensor de signo

Desplazador a la izquierda:

Para implementar la multiplicación por 4.

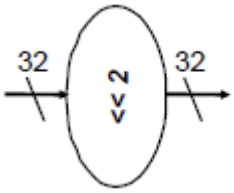

Figura 13 Desplazador a la izquierda

Banco de registros:

El MIPS dispone de un Banco de 32 registros, cuyo esquema se representa en Figura 14, cada uno de 32 bits de tamaño. La designación de los registros es $0, $1,....., $31.

Dado que las instrucciones de tipo R requieren el acceso simultáneo a 3 registros, la configuración de entradas y salidas del banco de registro serán:

- 2 salidas de datos de 32 bits.
- 1 entrada de datos de 32 bits.
- 3 entradas de 5 bits para la identificación de los registros.
- 1 entrada de control para habilitar la escritura sobre uno de los registros.
- 1 puerto de reloj (sólo determinante durante las operaciones de escritura, las de lectura son combinacionales)

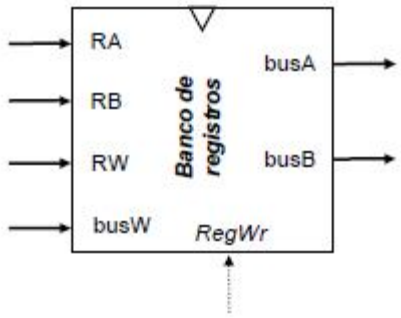

Figura 14 Banco de registros

Memoria:

La memoria debe tener un comportamiento idealizado. Para ello debe estar integrada dentro de la CPU, y debe contar con las siguientes características:

- Direccionable por bytes, pero capaz de aceptar/ofrecer 4 bytes por acceso
    - 1 entrada de dirección.
    - 1 salida de datos de 32 bits.
    - 1 entrada de datos de 32 bits (sólo en la de datos).
- Se supondrá que se comporta temporalmente como el banco de registros (síncronamente) y que tiene un tiempo de acceso menor que el tiempo de ciclo.
- Se supondrá dividida en dos para poder hacer dos accesos a memoria en el mismo ciclo:
    - Memoria de instrucciones, mostrada en la Figura 15.
    - Memoria de datos, mostrada en la Figura 16.

Figura 15 Memoria de instrucciones

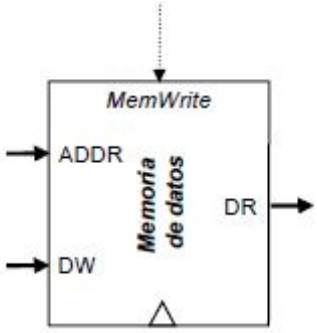

Figura 16 Memoria de datos

La memoria de datos tiene una entrada de control *MemWrite*, que permite escribir en ella si se encuentra está activa. En otro caso sólo leerá de las direcciones indicadas en *addr*.

Ruta de Datos

Ensamblamos todos los componentes anteriores, y obtenemos la siguiente ruta de datos mostrada a continuación en la Figura 17.

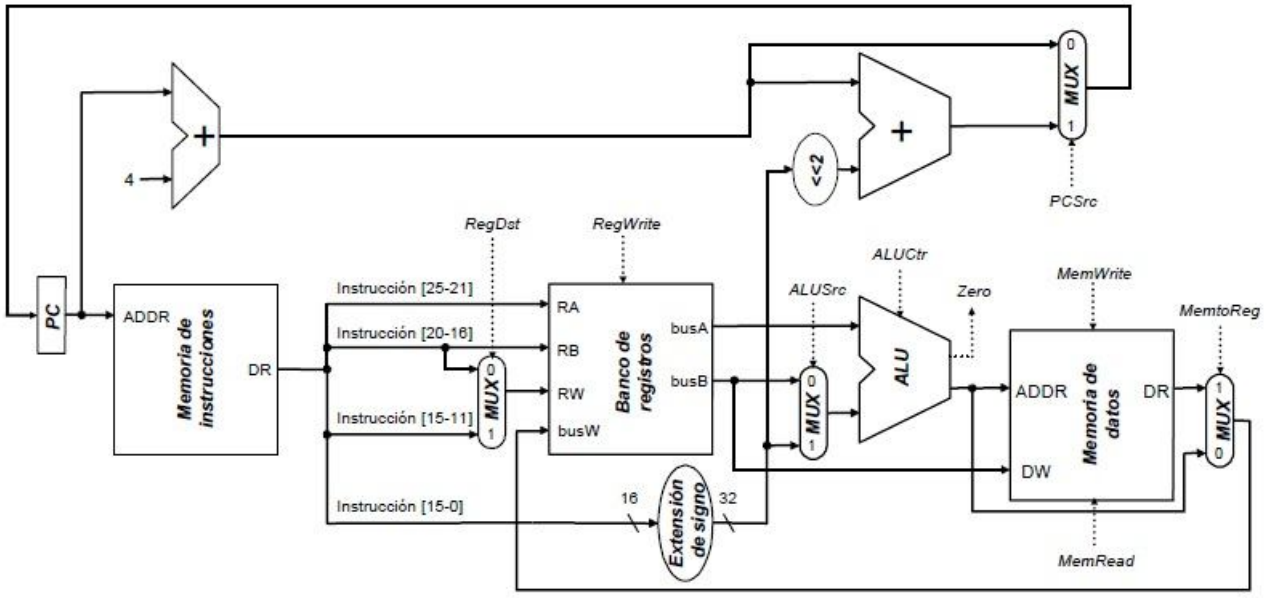

Figura 17 Ruta de datos monociclo

La ejecución monociclo ha obligado a utilizar cada recurso una sola vez por instrucción, de hecho se han tenido que duplicar algunos. También se han separado la memoria de datos y la de instrucciones.

Por otro lado, cuando un valor pueda provenir de varias fuentes el procesador se vale de multiplexores, que deciden de dónde se debe leer el dato.

El controlador, ilustrado en la Figura 18 es el encargado que todo funcione como es debido. Sus tareas son:

- Seleccionar las operaciones a realizar por los módulos multifunción.
- Controlar el flujo de datos, activando la entrada de selección de los multiplexores y la señal de carga de los registros

Todas las operaciones aritméticas comparten el mismo código de operación y durante su ejecución todas las señales generales de la ruta de datos son iguales. Por ello, utilizaremos:

- Un control principal para decodificar el campo de código de operación (op) y configurar globalmente la ruta de datos
- Un control local a la ALU que decodifique el campo de operación aritmética (funct) y seleccione la operación que debe realizar la ALU
- Adicionalmente en operaciones no aritméticas (lw, sw y beq) el control principal puede ordenar alguna operación a la ALU para calcular las DE o realizar comparaciones.

Utilizaremos la señal intermedia ALUop cuyo valor será:

- 00 en operaciones con acceso a memoria
- 01 en operaciones de salto
- 10 en operaciones aritméticas

Del mismo modo para controlar qué dirección debe cargar el PC se utilizará una señal intermedia Branch (activada durante la instrucción beq) a la que se hará la y-lógica con la señal Zero que genera la ALU.

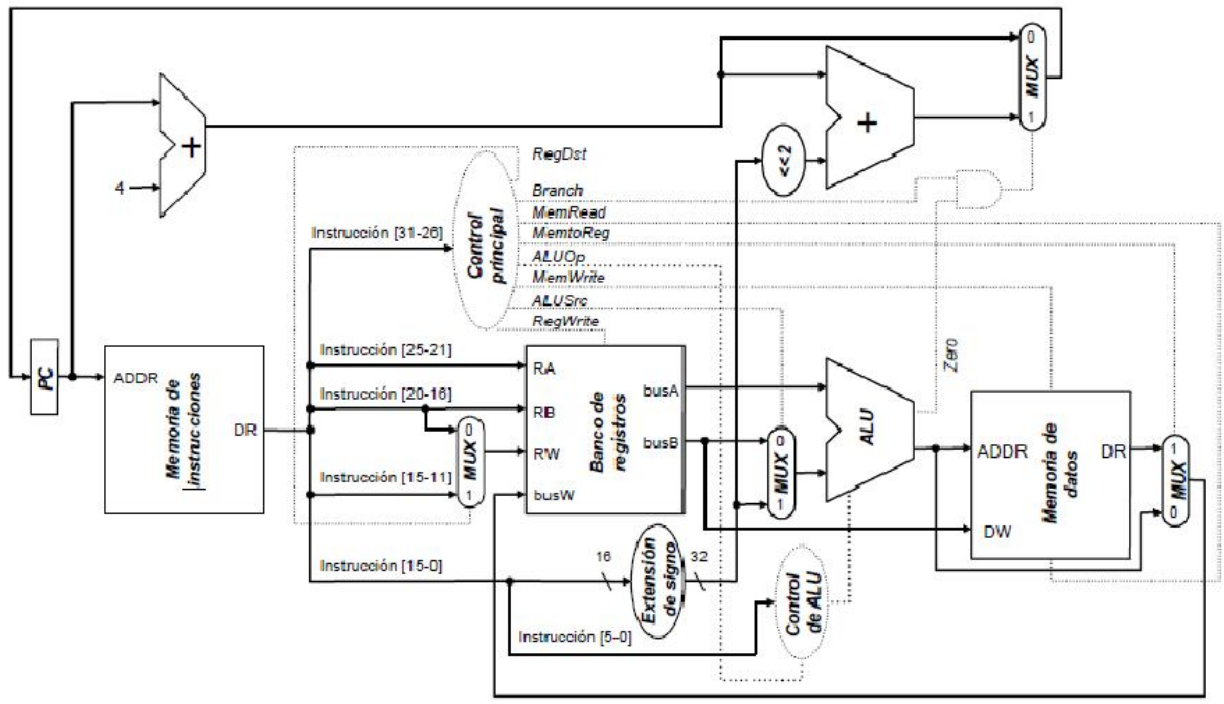

Figura 18 Ruta de datos y controlador

Por ejemplo, el siguiente cronograma de la Figura 19 representa la ejecución de la instrucción de carga: *lw*.

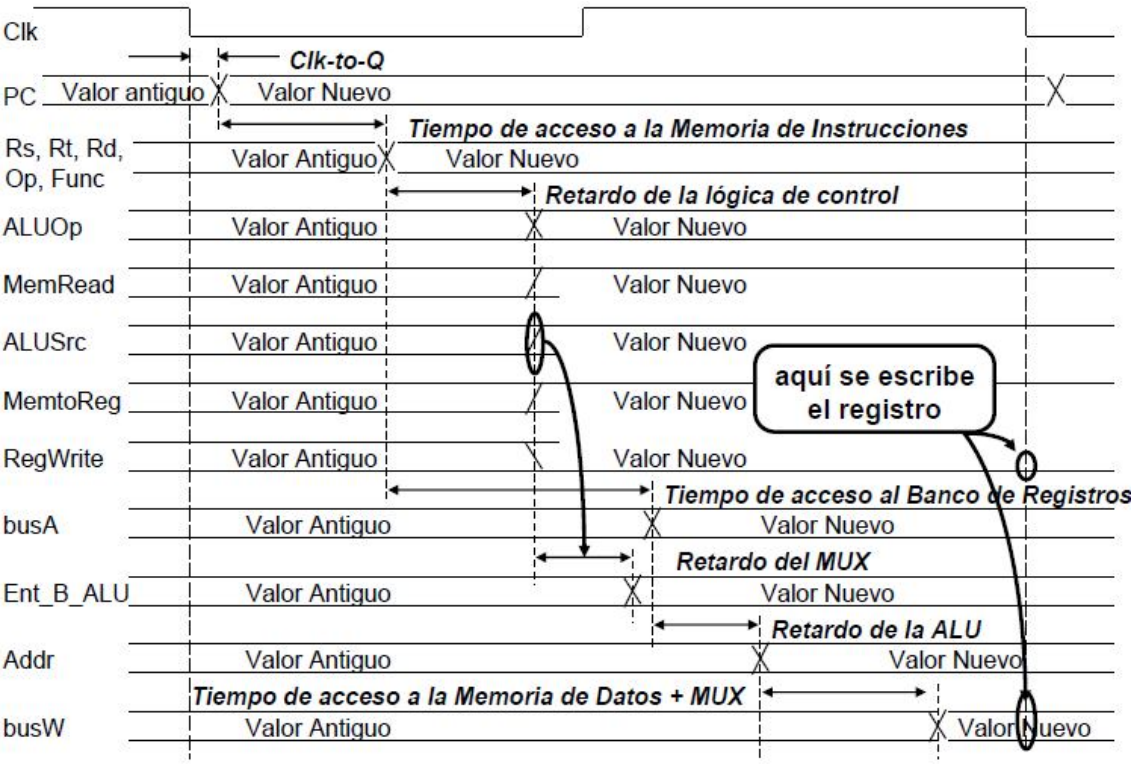

Figura 19 Cronograma instrucción lw

4. Implementación del procesador MIPS32 usando DEVS

4.1.- Introducción

DEVS, acrónimo de Discrete EVent system Specification, fue introducido a mediados de la década de los 70 por Bernard Zeigler. Se trata de un formalismo general, modular y jerárquico que sirve para el modelado y análisis de sistemas cuyo comportamiento pueda describirse mediante una secuencia de eventos discretos.

La simulación por eventos discretos es una técnica informática de modelado dinámico de sistemas que se caracteriza por un control en la variable del tiempo que permite avanzar a éste a intervalos variables en función de la ocurrencia de eventos en un tiempo futuro.Se permite representar cualquier sistema que tenga un número finito de cambios en un intervalo finito de tiempo.

La ejecución del sistema puede describirse mediante tablas de transición de estados. DEVS puede ser visto como una extensión de la máquina de estados de Moore, con las siguientes incorporaciones:

1. Se asocia un tiempo de vida a cada estado.

2. Para establecer una jerarquía agregamos la operación *coupling*.

Mediante las simulaciones que se harán con DEVS podremos realizar un estudio sobre el banco de registros del procesador MIPS32, con el objetivo de encontrar nuevas políticas que mejoren aquellos puntos menos eficientes. En nuestro caso, un estudio térmico sobre el banco de registros nos orientará a cómo reasignar los registros dentro del banco.

XDEVS

xDEVS es una plataforma de modelado y simulación DEVS implementada en JAVA. Está disponible como código abierto en sourceforge. Esta plataforma permite el modelado de componentes atómicos y acoplados. También cuenta con una extensa librería de componentes predefinida, principalmente orientadas a control de vehículos aéreos y otros elementos de ámbito militar.

También cuenta con algunos modelos diseñados para la optimización usando programación evolutiva. La simulación de modelos se realiza con Coordinadores, implementados para ejecutar simulaciones en tiempo virtual o en tiempo real.

4.2.- Descripción

Modelo DEVS atómico

Un modelo DEVS atómico (ver Figura 20) viene dado por la estructura:

$$M = <X, Y, S, \delta int, \delta ext, \lambda, t_a>$$

Donde

X: Conjunto de valores de entrada.

Y: Conjunto de valores de salida.

S: Conjunto de valores de estado.

δ_{int}: $S \rightarrow S$, función de transición interna. Define cómo cambia internamente un estado del sistema (cuando el tiempo transcurrido llega al tiempo de vida del estado)

δ_{ext}: $Q \times X \rightarrow S$**,** función de transición externa. Define como un evento de entrada cambia el estado del sistema. Donde

$$Q = \{(s,e) / s \in S, e \in [0, t_a(s)]\}$$

λ: $S \rightarrow Y$, función de salida. Define como un estado del sistema genera un evento de salida (cuando el tiempo transcurrido llega al tiempo de vida del sistema).

t_a: $S \rightarrow \mathbb{R}^{+,0}$ Define la función de avance del tiempo, el cual es usado para definir el tiempo de vida del estado.

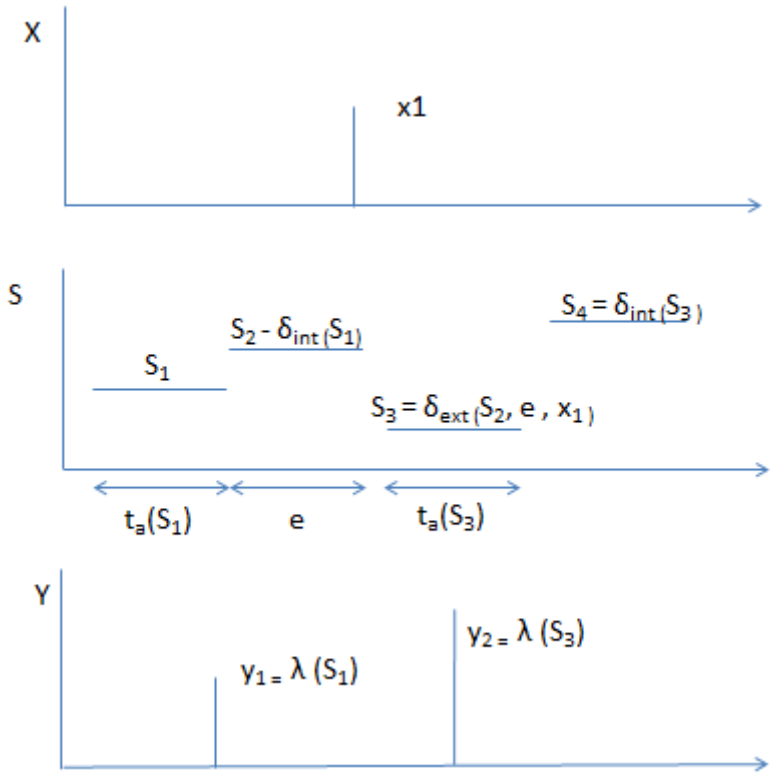

Figura 20 Modelo DEVS atómico

Los modelos se comunican entre sí a través de puertos. Los eventos de salida de un modelo se pueden convertir en eventos de entrada a otro modelo. Cada estado del modelo tiene un tiempo de vida definido por la función de duración (t_a). Cuando el tiempo transcurrido llega al tiempo de vida del estado, la transición interna se activa (δ_{int})y se produce un cambio interno de estado. Después de este cambio, el estado actual puede reflejarse en los puertos de salida.

Los valores de salida son enviados por la función de salida (λ), que debe ejecutarse después de la transición interna. En cualquier instante el modelo puede recibir eventos externos de entrada, cuando ocurre la función de transición externa (δ_{ext}) se activa y con la información del estado actual, los valores de entrada y la función de duración se realiza la transición a un nuevo estado, cuyo tiempo de vida se inicia.

Modelo DEVS acoplado

El modelo acoplado define cuales subcomponentes lo integran y cómo están conectados los unos con los otros.

Un modelo DEVS **acoplado** viene dado por la estructura:

$$M = <X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select>$$

Donde

X: Conjunto de eventos de entrada.

Y: Conjunto de eventos de salida.

D: Índice de subcomponentes ($\forall i \in D$).

M_i: Conjunto de subcomponentes donde para cada $i \in D$, $M_i$puede ser bien un modelo DEVS atómico o acoplado.

I_i: Influencias del modelo i ($\forall j \in I_i$)

Z_{ij}: $Y_i \rightarrow X_j$, función de translación de i a j

select: En caso de "empate", criterio de selección de un evento entre un conjunto de eventos simultáneos.

En el acoplamiento modular los eventos de salida de un modelo DEVS se convierten en eventos de entrada de otro. A su vez, el acoplamiento de modelos atómicos DEVS define un modelo atómico equivalente.

Cada componente se identifica como un índice. La función de translación usa un índice de influencias creado para cada modelo (M_i). La función define qué salidas del modelo M_i están conectadas a entradas del modelo M_j. Cuando dos submodelos tienen eventos simultáneos, la función de selección decide cual debe ser actividad primero.

Ejemplo DEVS

Bajo el formalismo DEVS, atomic DEVS captura el comportamiento del sistema mientras coupled DEVS describe la estructura del sistema.

Con el fin de ilustrar el funcionamiento de un modelo DEVS atómico, vamos a desarrollar el siguiente ejemplo:

Como vemos en la Figura 21 para un jugador de ping-pong, la entrada es el evento ?*recibir,* y la salida es el evento *!enviar.* Cada jugador, A, B, tiene sus respectivos estados *Enviar, Esperar.* El estado *Enviar* tarda 0'1segundos en devolver la pelota, lo que representa el evento de salida *!enviar*, mientras *Esperar* tardará hasta que el jugador reciba la pelota que es un evento de entrada ?*recibir.*

La estructura del juego de ping-pong conecta los dos jugadores: La salida del jugador A *!enviar* es transmitida a la entrada del jugador B ?*recibir*, y viceversa.

El modelo atómico para el jugador de ping-pong es como sigue:

X: {?enviar}

Y: {!recibir}

S: $\{(d,\sigma) \mid d \in \{\text{Enviar, Esperar}\}, \sigma \in [0,\infty]\}$ y S_0= (Enviar, 0'1)

$\delta_{int(Enviar,\sigma) = (Esperar, \infty)}$, $\delta_{int(Esperar,\sigma) = (Enviar, 0'1)}$

δ_{ext}(((Esperar, σ), e), ?recibir) = (Enviar, 0'1)

δ_{ext}(((Enviar, σ), e), ?recibir) = (Enviar, e), e tiempo transcurrido desde último evento.

λ(Enviar, σ) = !enviar

$t_{a:}\sigma$ ($\forall s \in S$)

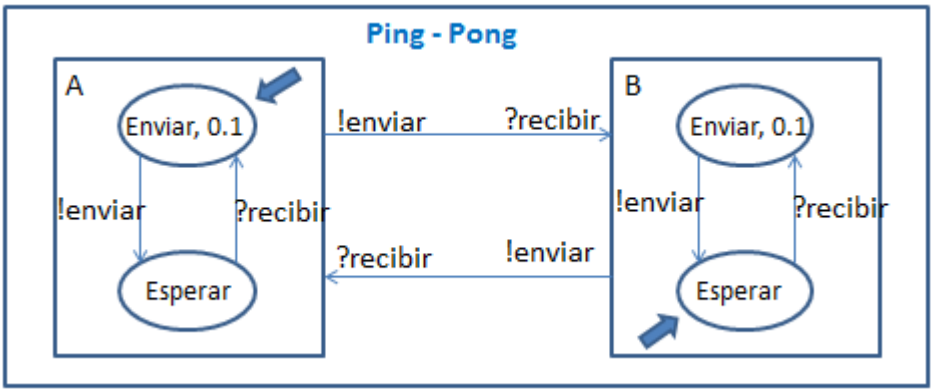

Figura 21 Ejemplo DEVS Ping-Pong

Simulación de sistemas DEVS

El algoritmo de simulación de los modelos DEVS considera dos cuestiones: (1) sincronización del tiempo y (2) propagación de mensajes. La *sincronización del tiempo* en DEVS conlleva controlar todos los modelos de manera que todos tengan su tiempo actual idéntico. Sin embargo, el algoritmo hace que el tiempo actual salte al tiempo más urgente cuando un evento está programado para ejecutar su transición interna. La *propagación de mensajes* en DEVS conlleva transmitir un mensaje, el cual puede ser un evento de entrada o de salida, a sus *couplings* asociados.

Los modelos DEVS pueden ser simulados de manera muy sencilla y eficiente. Para un simulador genérico de modelos DEVS el siguiente algoritmo puede usarse:

1. Para cada atómico d∈ N (modelo acoplado), calculamos el tiempo del próximo evento interno en el atómico d.
2. Avanzamos el tiempo total de la simulación hasta el tiempo del próximo evento calculado anteriormente.
3. Propagamos el evento de salida a todos los modelos atómicos conectados al puerto de salida de dicho evento y ejecutamos las funciones de transición externas correspondientes.
4. Repetimos desde el paso 1.

Para implementar este algoritmo en un programa debe usarse una estructura jerárquica del modelo.

Cada modelo atómico tiene asociado un simulador DEVS, mientras que cada modelo acoplado tiene asociado un coordinador DEVS.

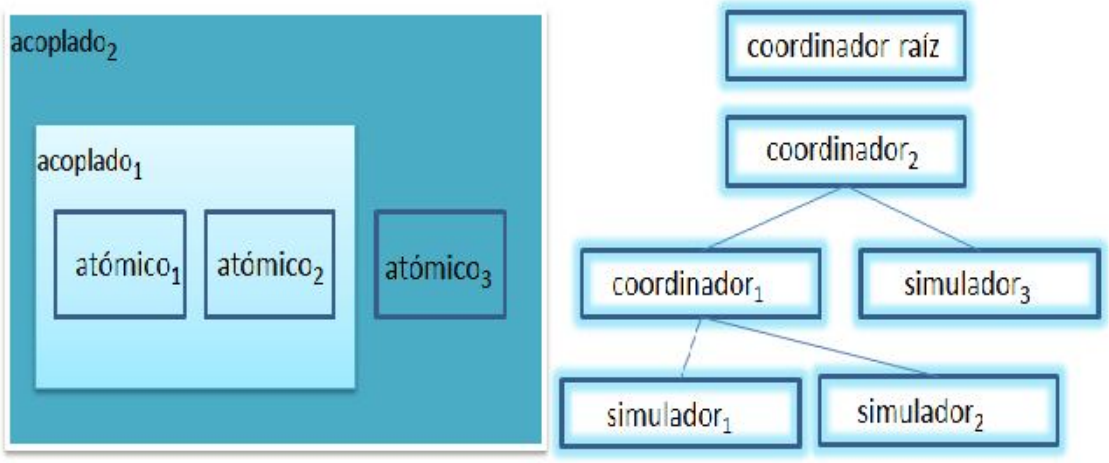

Figura 22 Sistema DEVS

El proyecto base sobre el que trabajamos se desarrolla sobre xDEVS por estar desarrollado en el Departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid. Además, xDEVS ha sido validado con la simulación de varios modelos publicados en congresos y revistas de ámbito internacional.

4.3.- Instrumentación del Banco de Registros usando DEVS.

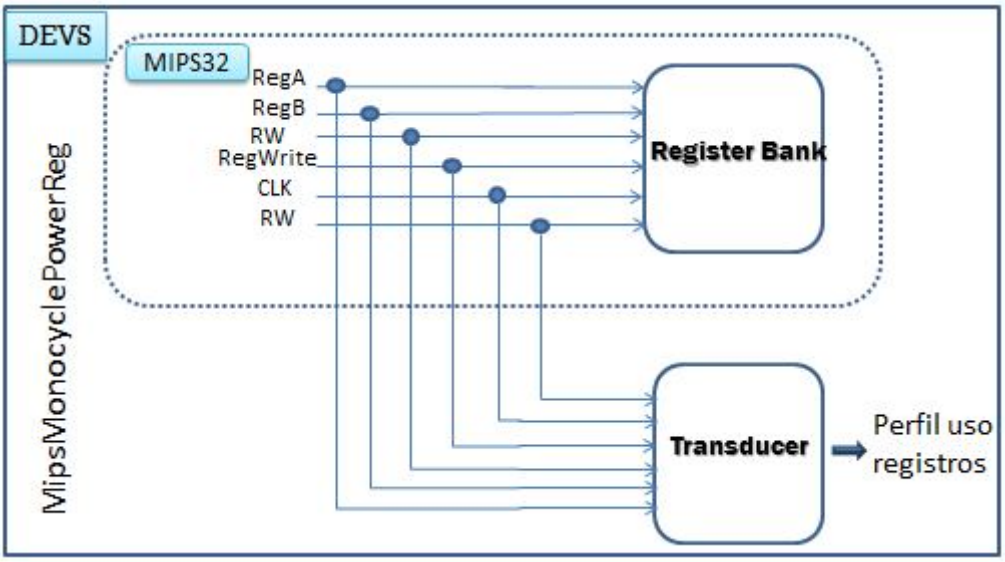

Figura 23 Instrumentación del banco de registros usando DEVS. Conexión transductor

Partimos de la base de un simulador MIPS32 monociclo implementado en su totalidad usando DEVS. Añadimos dos clases *ad hoc* que nos permitan obtener una traza de los registros:

- Empezamos creando una clase que extiende al MIPS monociclo ya existente, pero limitándonos a las conexiones en las que interviene el banco de registros, como vemos en la figura. Las conexiones se realizan entre un transductor –TransducerPowerReg- y el banco de registros de manera que las entradas al banco de registros son también las entradas al transductor con la instrucción de acoplamiento *addCoupling ()*. Las entradas que se conectan son: RegA (registro A), RegB (registro B), RW (registro destino) sobre el que se escribirá cuando se active la señal RegWrite, y CLK. La clase es MipsMonocyclePowerReg y extiende a MipsMonocycle.

Implementación MipsMonocyclePowerReg

```
public class MipsMonocyclePowerReg extends MipsMonocycle {

   protected TransducerPowerReg transducer;

   public MipsMonocyclePowerReg(String name, String file Path)throws
   FileNotFoundException {
      super(name, file Path);
      transducer =new TransducerPowerReg("Transducer",Registers.WRITE_MODE.PERIOD);
      super.addComponent(transducer);
      super.addCoupling(insMem, InstructionsMemory.outStopName, transducer,
          TransducerPowerReg.inStopName);
      super.addCoupling(clock, Clock.outName, transducer, TransducerPowerReg.inClkName);
      super.addCoupling(insNode, InsNode.outOut2521Name, transducer,
          TransducerPowerReg.inRAName);
      super.addCoupling(insNode, InsNode.outOut2016Name, transducer,
          TransducerPowerReg.inRBName);
      super.addCoupling(muxRegDst, Mux2to1.outOutName, transducer,
          TransducerPowerReg.inRWName);
      super.addCoupling(ctrl, ControladorMono.outRegWriteName, transducer,
          TransducerPowerReg.inRegWriteName);
   }
   public static void main(String[] args) {
      if(args.length!=2) {
         System.out.println
              ("java -jar MipsTemp.jar <AssemblyFilePath> <NumPicoSeconds e.g. 0.001e12>");
         args = new String[2];
         args[0] = "test" + File.separator + "game_of_life_o1.dis";
         args[1] = "0.001e12";
         return;
      }
      try {
         // Diferentes pruebas para ejecutar:
         benchmark(args[0], Double.valueOf(args[1]));
      } catch (FileNotFoundException ex) {
         Logger.getLogger(MipsMonocyclePowerReg.class.getName()).log
         (Level.SEVERE, null, ex);
      }
   }
public static void benchmark(String file Path, Double numSeconds) throws
          FileNotFoundException {
      // Benchmark1: Función que calcula el producto escalar de dos vectores
      MipsMonocyclePowerReg mips = new MipsMonocyclePowerReg("MIPS", filePath);
      CoordinatorLogger coordinator = new CoordinatorLogger(mips);
      coordinator. simulate(numSeconds);
      mips.transducer.logReport();
   }
}
```

- La clase que implementa el transductor es TransducerPowerReg y extiende a Atomic.
  Como salida obtenemos un *log* que va guardando a modo de historial las lecturas y escrituras que se efectúan sobre cada registro de manera que obtenemos un perfil de los registros. Este perfil hará posible la extracción de información como la potencia que consume cada registro. Para el cálculo de la potencia, hemos calculado previamente la energía usando los siguientes valores:

  Energía escritura: 0,1408576nJ

  Energía lectura: 0,25455 nJ

  Según la operación realizada, se suma el valor de energía de lectura o escritura respectivamente al registro afectado.

Implementación TransducerPowerReg

```
public class TransducerPowerReg extends Atomic {

   private static final Logger logger =
=Logger.getLogger(TransducerPowerReg.class.getName());

   public static final double ENERGY_WRITE = 0.1408576 * 10E-9;
   public static final double ENERGY_READ = 0.25455 * 10E-9;
   public static final String inStopName = "Stop";
   public static final String inClkName = "CLK";
   public static final String inRegWriteName = "RegWrite";
   public static final String inRAName = "RA";
   public static final String inRBName = "RB";
   public static final String inRWName = "RW";
   protected Port inStop = new Port(TransducerPowerReg.inStopName);
   protected Port<Integer> inClk = new Port<Integer>(TransducerPowerReg.inClkName);
   protected Port<Integer> inRegWrite = new Port<Integer>(inRegWriteName);
   protected Port<Integer> inRA = new Port<Integer>(inRAName);
   protected Port<Integer> inRB = new Port<Integer>(inRBName);
   protected Port<Integer> inRW = new Port<Integer>(inRWName);
   protected double time = 0.0;
   protected WRITE_MODE writeMode;
   protected Integer valueAtRA = null;
   protected Integer valueAtRB = null;
   protected Integer valueAtRW = null;
   protected Integer valueAtRegWrite = null;
   protected Integer valueAtCLK = null;
   protected double[] energyVector;

   public TransducerPowerReg(String name, WRITE_MODE writeMode) {
      super(name);
      super.addInport(inStop);
```

```
        super.addInport(inClk);
        super.addInport(inRegWrite);
        super.addInport(inRA);
        super.addInport(inRB);
        super.addInport(inRW);
        this.writeMode = writeMode;
        this.energyVector = new double[32];
        super.passivate();
    }

    @Override
    public void deltint() {
        super.passivate();
    }

    @Override
    public void deltext(double e) {
        super.resume(e);
        time += e;
        Integer tempValueAtRA = inRA.getValue();


        if (tempValueAtRA != null && tempValueAtRA != valueAtRA) {
            valueAtRA = tempValueAtRA;
            energyVector[valueAtRA] += ENERGY_READ;
            logger.log(Level.FINE, time + " RA " + valueAtRA);
        }
        Integer tempValueAtRB = inRB.getValue();
        if (tempValueAtRB != null && tempValueAtRB != valueAtRB) {
            valueAtRB = tempValueAtRB;
            energyVector[valueAtRB] += ENERGY_READ;
            logger.log(Level.FINE, time + " RB " + valueAtRB);
        }
        Integer tempValueAtRW = inRW.getValue();
        if (tempValueAtRW != null) {
            valueAtRW = tempValueAtRW;
            energyVector[valueAtRW] += ENERGY_READ;
            logger.log(Level.FINE, time + " RW " + valueAtRW);
        }
        Integer tempValueAtRegWrite = inRegWrite.getValue();
        if (tempValueAtRegWrite != null) {
            valueAtRegWrite = tempValueAtRegWrite;
        }
       // Ahora el reloj, que gobierna la escritura:
        Integer tempValueAtCLK = inClk.getValue();
        if (tempValueAtCLK != null) {
            boolean write = false;
            if (writeMode == WRITE_MODE.SEMI_PERIOD) {
                // Se escribe si es un paso de 0 a 1
                if (valueAtRegWrite != null && valueAtRegWrite == 1 && valueAtCLK != null &&
                  valueAtCLK == 0 && tempValueAtCLK == 1) {
```

```
                write = true;
            }
        } else if (writeMode == WRITE_MODE.PERIOD) {
            // Se escribe si es un paso de 1 a 0
            if (valueAtRegWrite != null && valueAtRegWrite == 1 && valueAtCLK != null &&
              valueAtCLK == 1 && tempValueAtCLK == 0) {
                write = true;
            }
        }


        if (write) {
            energyVector[valueAtRW] += ENERGY_WRITE;
            logger.log(Level.FINE, time + " RX " + valueAtRW);
        }
      valueAtCLK = tempValueAtCLK;
      }
      if(!inStop.isEmpty()) {
         super.holdIn("finish", 0.0);
      }
   }

   @Override
   public void lambda() {
   }

   public void logReport() {
      for(int i=0; i<energyVector.length; ++i) {
         logger.log(Level.INFO, "Register #" + i + " = " + energyVector[i]/(time*Math.pow(10,-
12)) + " W");
      }
   }
}
```

Ejecutamos los benchmarks seleccionados para poder extraer resultados concluyentes. Cada benchmark nos proporciona un perfil del uso de los registros con el que calculamos la densidad de potencia de cada cual y procedemos al estudio térmico.

Ejemplo Banco de registros

A continuación definimos el elemento principal del procesador MIPS sobre el que trabajamos, implementado con DEVS

Banco de Registros

El procesador cuenta con un banco de 32 registros. Es un dispositivo de lectura/escritura.

Banco de registros = < X, Y, S, δ_{int}, δ_{ext}, λ, ta>

Puertos de entrada:{ CLK, RegWrite, RA, RB, RW, busW}

Puertos de salida = {busA, busB}

X ∈ {{0,1}, {0,1},$\mathbb{Z}^+$,$\mathbb{Z}^+$,$\mathbb{Z}^+$,$\mathbb{Z}$}

S = {σ∈$\mathbb{R}_0^+$, *data*∈ $\mathbb{Z}^n$, *tw*∈ $\mathbb{R}_0^+$es tiempo de retardo en escritura, *tr*∈ $\mathbb{R}_0^+$es tiempo de retardo en lectura}

S_0 = {∞, 0^n, *tw*, *tr*} es el estado en el instante inicial.

Y ∈ {$\mathbb{Z}$, $\mathbb{Z}$}

δ_{int} ()

passivate()

δ_{ext} (X', e)

X = X'

si (RA or RB) entonces

σ = *tr*

si (RegWrite = 1, CLK = ↓) entonces

data [RW] = busW, σ = *tw*

λ ()

5. Algoritmos genéticos

Para conseguir una reasignación óptima de los registros en el banco de registros, de manera que la temperatura total sea mínima, hemos aplicado un algoritmo genético cuyo objetivo es colocar cada uno de los registros cálidos lo más separados entre sí.

5.1. Introducción a MOEA

Los algoritmos genéticos forman un subconjunto de la computación evolutiva que abarca problemas combinatorios de optimización. La computación evolutiva se basa en procesos iterativos tales como el crecimiento y desarrollo de una población que evoluciona iterativamente mediante una búsqueda aleatoria guiada para conseguir el fin deseado.

Los procesos de búsqueda guiada están inspirados en los mecanismos biológicos de la evolución tales como reproducción, mutación, selección natural y supervivencia del más "fuerte". Las soluciones candidatas al problema de optimización hacen el papel de individuos en una población, y la función de idoneidad determina la adaptación al medio donde viven las soluciones y cuyo valor es lo que determina si un individuo tiene mayor o menor probabilidad de ser seleccionado. La evolución de la población se lleva a cabo después de la aplicación reiterada de estos mecanismos.

MOEAs trata de heurísticas estocásticas de optimización donde la exploración del espacio de soluciones de un determinado problema es llevado a cabo según la teoría de la evolución antes mencionada.

En este estudio nos centramos en el algoritmo evolutivo multi-objetivo "Non-dominated Sorting Genetic Algorithm" (NSGA-II) que se basa en una organización jerárquica de los individuos. . Para más información sobre el funcionamiento de NSGA-II se recomienda ver (Deb, 2002).

En nuestro caso se tendrán en cuenta sólo dos dimensiones ya que no se aplica a procesadores sino únicamente al banco de registros.

Tomaremos cada una de las configuraciones del banco de registros como un cromosoma, formado por 32 alelos cada uno. Antes de comenzar con el algoritmo se debe comprobar que cada uno de los cromosomas es correcto, esto es que esté formado por 32 alelos, o registros, y que estos sean distintos entre sí. Por ejemplo, el siguiente cromosoma de la Figura 24 sería considerado como correcto.

R4	R18	R30	R9	R16	R11	R27	R8	R24	R15	R3	R21	R17	R2	R12	R25
R23	R10	R19	R0	R5	R28	R22	R31	R6	R13	R14	R26	R1	R20	R29	R7

Figura 24 Ejemplo de cromosoma

Cada uno de los registros R_i estará determinado por sus variables:

- x_i: posición del registro en el eje de abscisas.
- y_i: posición del registro en el eje de ordenadas.

Este valor será relativo dentro de cada cromosoma, tomando el (0,0) como la esquina inferior izquierda. En el ejemplo anterior, el registro R23 tendrá como variables (0, 0) mientras que los valores de las coordenadas para el registro R10 serán (x_{23}, 0). Hay que recordar, por otro lado, que en el banco de registros, todos los registros tienen las mismas dimensiones de anchura (w), longitud (l).

5.2. Ejemplo

Esta metodología, como todo algoritmo genético, está formada por tres fases:

- Selección
- Cruce
- Mutación.

Para explicar cada una de ellas se acompañará de un ejemplo, suponemos para este caso, y por comodidad, un banco de registros de 8 registros.

Selección

Se toman al azar dos cromosomas de toda la población, cuando acabe todo el proceso se habrán creado dos hijos y se unirán al resto. Antes de comenzar de nuevo con la siguiente selección, se conservará una población del mismo tamaño que la inicial con los N mejores individuos.

Para nuestro ejemplo, suponemos que se han seleccionado los siguientes cromosomas, como se ilustra en la Figura 25 y Figura 26.

R1	R3	R6	R2	R0	R5	R4	R7

Figura 25 Ejemplo cromosoma 1

R4	R2	R5	R1	R6	R0	R7	R3

Figura 26 Ejemplo cromosoma 2

Cruce

Se parte del primer alelo del cromosoma 1, esto es R1, y se busca en el cromosoma 2.

R1	R3	R6	R2	R0	R5	R4	R7

R4	R2	R5	R1	R6	R0	R7	R3

Desde R1 en el cromosoma 2, nos movemos al alelo que tenga en el cromosoma 1 en la misma posición, el registro R2 y se repite el proceso.

R1	R3	R6	R2	R0	R5	R4	R7

R4	R2	R5	R1	R6	R0	R7	R3

Se busca R2 en el cromosoma 2 y nos desplazamos al alelo del cromosoma 1, que se encuentre en la misma posición que R2 en el cromosoma 2.

R1	R3	R6	R2	R0	R5	R4	R7

R4	R2	R5	R1	R6	R0	R7	R3

Buscamos el alelo R3 en el cromosoma 2. En alelo R7 del cromosoma 1 se encuentra en la misma posición que R3 en el cromosoma 2.

R1	R3	R6	R2	R0	R5	R4	R7

R4	R2	R5	R1	R6	R0	R7	R3

Repetimos el proceso con R7.

R1	R3	R6	R2	R0	R5	R4	R7

R4	R2	R5	R1	R6	R0	R7	R3

Seguimos con el alelo R4.

R1	R3	R6	R2	R0	R5	R4	R7

R4	R2	R5	R1	R6	R0	R7	R3

Lo buscamos en el cromosoma 2.

R1	R3	R6	R2	R0	R5	R4	R7

R4	R2	R5	R1	R6	R0	R7	R3

Observamos que cuando lleguemos a R4 en el cromosoma 2 no nos podemos desplazar al que tiene en la misma posición en el otro cromosoma porque ya lo hemos "tocado".

En el cromosoma 1 se han dejado sin visitar los registros R6, R4 y R7. Repetimos el cruce cambiando los cromosomas de orden. Vemos que los alelos sin visitar del cromosoma 2 están en la misma posición que los alelos sin visitar en el cromosoma 1. Para el cromosoma 2, los alelos sin visitar son R5, R6 y R0.

R1	R3	R6	R2	R0	R5	R4	R7

R4	R2	R5	R1	R6	R0	R7	R3

El cruce se realizará entre los registros marcados en azul, que son los que no hemos visitado en el proceso anterior. Por lo tanto, los cromosomas quedarán de la siguiente manera:

R1	R3	R5	R2	R6	R0	R4	R7

R4	R2	R6	R1	R0	R5	R7	R3

Mutación

En este caso, como no podemos variar las dimensiones de cada uno de los registros, la mutación consistirá en intercambiar pares de alelos según la probabilidad en ese momento, según una determinada probabilidad. Se puede decidir este valor, por ejemplo, como:

Probabilidad = 1/número de registros.

Siguiendo el ejemplo anterior, suponemos que se intercambian los alelos 2 y 7 del cromosoma 1 y los alelos 1 y 4 del cromosoma 2.

R1	R4	R5	R2	R6	R0	R3	R7

R1	R2	R6	R4	R0	R5	R7	R3

La función fitness

Se trata de una función evaluadora de la calidad de un individuo como solución a nuestro problema. Permite la ordenación de los individuos de la población en cuanto a bondad de los mismos.

En nuestro caso particular, cada cromosoma representa el orden en que los registros están colocados en el banco de registros.

Tenemos dos objetivos, que se describen a continuación:

1. Restricciones topológicas (J1): Debemos tener en cuenta que ningún registro se sale del área del banco de registros, también hay que cuidar que no existan registros superpuestos. Si estas restricciones se cumplen, el objetivo J1 tendrá valor 0.0.

2. Temperatura (J2): Este último objetivo es una medida del impacto térmico. Para calcularlo debemos tener en cuenta, no sólo el consumo de potencia de cada uno de los registros, sino también el efecto que tienen cada uno de los registros con sus vecinos.

El valor de estos últimos objetivos, se calcula de la siguiente forma:

$$\sum_{i<j\in 1..32} \frac{(dpi * dpj)}{dij}$$

Siendo *dp* la densidad de potencia de cada registro y d_{ij} la distancia euclídea entre los registros *i* y *j*

6. Estudio Térmico

6.1 Análisis térmico

La alta utilización de recursos y la potencia consumida en consecuencia es la fuente de los problemas a los que se enfrentan diseñadores de microprocesadores. En la etapa inicial de diseño, entender las características de la potencia y sus ramificaciones es esencial dado que las decisiones arquitectónicas pueden tener un gran impacto en la eficiencia global. Además, las flaquezas a nivel arquitectónico son demasiado difíciles de corregir en etapas de diseño posteriores.

Altas densidades de potencia conllevan altas temperaturas y esto perjudica la fiabilidad de los componentes e incrementa la fuga de potencia. Optimizar la potencia disipada depende de un modelado eficaz y eficiente que cubra distintas disciplinas y niveles, desde la física del dispositivo hasta el diseño arquitectónico.

La temperatura que adquiere el banco de registros de un procesador, y a mayor escala la superficie del chip, está ligada al uso de los registros. Si conseguimos optimizar el uso de determinados registros mediante políticas de asignación eficientes, lograremos minimizar la energía disipada. De esta manera evitamos las repercusiones negativas que acarrea el calentamiento de estos elementos de almacenamiento, tales como el aumento de consumo de potencia, el deterioro del procesador, pues de lo contrario sería necesario un enfriamiento del chip, lo que aumentaría costes.

6.2 Impacto del compilador en la temperatura

La fase de asignación de registros de un compilador también tiene un impacto sobre la temperatura del sistema. Pueden provocar los siguientes errores:

- Acceso concurrente al mismo registro. Ocurre cuando el compilador asigna repetidamente distintos registros lógicos a un mismo registro físico, y por consiguiente este registro sufrirá un mayor calentamiento.

- Acceso a registros localizados en una misma área. Ocurre cuando los registros asignados se encuentran próximos entre sí en el espacio físico del chip y esto provoca que se concentre el calor en un determinado área. Por otra parte el flujo de energía térmica va asociado a una diferencia de temperaturas y en este caso se produce una conducción de energía térmica por gradiente de temperatura, de manera que se calientan los registros adyacentes al usado.

Por todo esto, encontrar una reubicación de los registros supone un incremento en la fiabilidad de los componentes.

Por último, mediante benchmarks, pasamos al paso de la verificación de la reducción de temperaturas alcanzadas con la nueva reasignación.

6.3 Perfiles térmicos

Los perfiles térmicos dependen de la distribución temporal y espacial de la potencia. A continuación necesitamos cuantificar el impacto térmico bajo estudio. Estudiamos la energía consumida por los registros del banco para un conjunto de 3 benchmarks que siguen distintos patrones de accesos al banco de registros. Unos derivan en el uso prolongado de un conjunto reducido de registros, y otros en un uso más disperso de los 32 registros disponibles.

El banco de registros sobre objeto de nuestro estudio, es una matriz de *32 x 1.* Cada celda básica corresponde a un registro y sus dimensiones serán:

Ancho = 2 celdas Largo = 6 celdas

Dimensiones celdas: 50µm x 50µm

Temperatura

El análisis térmico sobre un circuito integrado es la simulación de la transferencia de calor a través de material heterogéneo entre productores de calor (e.g. transistores), y consumidores de calor (e.g. *heatsinks* adjuntados a los circuitos integrados).

Descomponemos el chip en elementos térmicos discretos, que pueden ser de formas y tamaños variables. Los elementos térmicos contiguos interactúan vía difusión de calor. Cada elemento tiene disipación de potencia, temperatura, capacitancia térmica y resistencia térmica a los elementos contiguos.

La siguiente ecuación gobierna la conducción térmica en chips:

$$C\frac{dT(t)}{dt} + AT(t) = Pu(t)$$

Donde la matriz de capacitancia térmica, C, es un matriz diagonal de *N x N.* La matriz de conducción térmica, A, es una matriz dispersa *N x N* celdas, $T(t)$ y $P(t)$ son vectores de temperatura y potencia *N x 1, y* $u(t)$es la función de paso de tiempo.

El análisis térmico en estado de equilibrio está caracterizado por la distribución de temperatura cuando el flujo de calor no varía con el tiempo. Por tanto, para un análisis en estado estacionario el término a la izquierda que expresa variación de temperatura en función del tiempo se anula. Así:

$$AT(t) = P \rightarrow T = A^{-1}P$$

De esta manera, dada la matriz A de resistencia térmica y el vector de potencia P, la tarea principal del análisis térmico es invertir A. La complejidad computacional de ese análisis esta, por ende, determinado por el tamaño de A.

Energía

Previo al cálculo de la potencia, necesitamos hallar la energía consumida por cada registro. Esta energía es un factor que depende del número de accesos al registro en cuestión. Este cálculo se hace sobre todos los registros obteniendo así el total de accesos de lectura y escritura sobre cada cual.

Utilizamos un modelo (Qadri, 2009) que describe la energía consumida como sigue:

Donde Er y Ew son la energía consumida por accesos de lectura y escritura respectivamente. Dado que en este estudio nos centramos en la energía consumida por operaciones sobre el banco de registros, únicamente consideraremos aquella energía que conllevan lecturas y escrituras.

Los valores de Er y Ew (Yeager, 1996) se definen:

$$Er = Nr \cdot Edr$$
$$Ew = Nw \cdot Edw$$

Nr, Nw: Número de lecturas y escrituras.

Edr, Ewr: Energía consumida por cada operación.

$$Edw = 0.1408576^{-9}\ \text{J}$$

$$Edr = 0.25455^{-9}\ \text{J}$$

El número de lecturas y escrituras es obtenido mediante la ejecución del simulador MIPS sobre DEVS.

7. Interfaz gráfica para la visualización de registros

La implementación del estudio térmico se ha realizado mediante un programa en Matlab, el cual recibe como entrada el ejemplo que se desea ejecutar con su distribución correspondiente y obtenemos como resultados la temperatura máxima, la temperatura media y el valor del gradiente.

El problema de dar solo unos valores numéricos, es que no sabemos qué registros alcanzan una temperatura mayor y qué registros una menor. Este dato nos es muy interesante dado que el objetivo de este proyecto es el de reasignar los registros del Banco de registros de manera que los registros cálidos estén separados entre sí.

Se pensó que la mejor manera de representar estos resultados era de una manera gráfica. Por una parte porque así se puede conocer aproximadamente el calor desprendido por cada uno de los registros que componen el Banco de registros del procesador, y por otro lado entendemos que siempre es más sencillo visualizar unos resultados de manera gráfica que leyendo unos números que, a priori, nos pueden resultar confusos.

Para la interfaz gráfica del Banco de registros se utilizará un código de colores bastante simple e intuitivo: azul para los registros fríos y rojo para los cálidos. La organización del Banco de registros es tal como imaginamos, esto es un rectángulo situado en los ejes X e Y, dividido en tantas celdas como registros lo componen estando éstas distribuidas según la organización correspondiente en cada momento (32x2, 16x2 o 8x4)

A continuación, se explicarán todos los pasos que han sido necesarios para realizar el estudio que centra este proyecto, y los programas utilizados para ello.

Se parte de un archivo en formato *XML* que representa un ejemplo de ejecución para una configuración del Banco de registros concreta. Este archivo contiene, además del formato del Banco y el tamaño de celda, la posición en el eje de coordenadas de cada uno de los 32 registros junto con el valor de la densidad de potencia (dp), la cual se expresa como:

$$dp = \text{potencia del registro i} / \text{área del registro i}$$

Por ejemplo, a continuación se muestra el archivo correspondiente al "Juego de la vida" siendo la configuración del Banco de registros de 8 filas por 4 columnas (8x4).

<? xml version="1.0" encoding="UTF-8”?>

<Floorplan Version="3.1" CellSize="300" Length="24" Width="16" NumLayers="1" NumPowerProfiles="1">

<Blocks>

<Block id="0" name="R0" type="0" xMin="0" x="0" xMax="0" yMin="0" y="0" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="753813.4231"/>

<Block id="1" name="R1" type="0" xMin="0" x="0" xMax="0" yMin="0" y="2" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="2" name="R2" type="0" xMin="0" x="0" xMax="0" yMin="0" y="4" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="3826845.623"/>

<Block id="3" name="R3" type="0" xMin="0" x="0" xMax="0" yMin="0" y="6" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="970138.363"/>

<Block id="4" name="R4" type="0" xMin="0" x="0" xMax="0" yMin="0" y="8" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="471322.0027"/>

<Block id="5" name="R5" type="0" xMin="0" x="0" xMax="0" yMin="0" y="10" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="604546.073"/>

<Block id="6" name="R6" type="0" xMin="0" x="0" xMax="0" yMin="0" y="12" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="263489.0873"/>

<Block id="7" name="R7" type="0" xMin="0" x="0" xMax="0" yMin="0" y="14" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="56071.7049"/>

<Block id="8" name="R8" type="0" xMin="0" x="6" xMax="0" yMin="0" y="0" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="335610.3079"/>

<Block id="9" name="R9" type="0" xMin="0" x="6" xMax="0" yMin="0" y="2" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="21315.0218"/>

<Block id="10" name="R10" type="0" xMin="0" x="6" xMax="0" yMin="0" y="4" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="13873.02819"/>

<Block id="11" name="R110" type="0" xMin="0" x="6" xMax="0" yMin="0" y="6" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="40203.61337"/>

<Block id="12" name="R12" type="0" xMin="0" x="6" xMax="0" yMin="0" y="8" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="99.27268806"/>

<Block id="13" name="R13" type="0" xMin="0" x="6" xMax="0" yMin="0" y="10" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="14" name="R14" type="0" xMin="0" x="6" xMax="0" yMin="0" y="12" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="15" name="R15" type="0" xMin="0" x="6" xMax="0" yMin="0" y="14" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="16" name="R16" type="0" xMin="0" x="12" xMax="0" yMin="0" y="0" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="17" name="R17" type="0" xMin="0" x="12" xMax="0" yMin="0" y="2" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="18" name="R18" type="0" xMin="0" x="12" xMax="0" yMin="0" y="4" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="19" name="R19" type="0" xMin="0" x="12" xMax="0" yMin="0" y="6" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="20" name="R20" type="0" xMin="0" x="12" xMax="0" yMin="0" y="8" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="21" name="R21" type="0" xMin="0" x="12" xMax="0" yMin="0" y="10" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="22" name="R22" type="0" xMin="0" x="12" xMax="0" yMin="0" y="12" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="23" name="R23" type="0" xMin="0" x="12" xMax="0" yMin="0" y="14" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="24" name="R24" type="0" xMin="0" x="18" xMax="0" yMin="0" y="0" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="25" name="R25" type="0" xMin="0" x="18" xMax="0" yMin="0" y="2" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="26" name="R26" type="0" xMin="0" x="18" xMax="0" yMin="0" y="4" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="27" name="R27" type="0" xMin="0" x="18" xMax="0" yMin="0" y="6" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="28" name="R28" type="0" xMin="0" x="18" xMax="0" yMin="0" y="8" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="29" name="R29" type="0" xMin="0" x="18" xMax="0" yMin="0" y="10" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="332919.0263"/>

<Block id="30" name="R30" type="0" xMin="0" x="18" xMax="0" yMin="0" y="12" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="0.0"/>

<Block id="31" name="R31" type="0" xMin="0" x="18" xMax="0" yMin="0" y="14" yMax="0" zMin="0" z="0" zMax="0" l="6" w="2" h="1" dp0="2.676755115"/>

</Blocks>

<Distances></Distances>

<Couplings></Couplings>

</Floorplan>

Cada registro viene representado por conjunto de variables que describen sus características. Este patrón se repite tantas veces como registros tiene el banco de registros.

Las variables “id” y “name” son los identificadores de cada uno de los registros. Por otro lado, observamos que las variables “x” y “y” van cambiando, esto es porque representan la posición del registro en el banco de registros, suponiendo que éste se encuentra apoyado en los ejes de ordenadas y abscisas.

Los valores de “w”, “l” y “h” nos indican las dimensiones de cada componente del banco de registros, como se observa no varía ya que todos los registros deben ser del mismo tamaño.

Por último, nos interesa conocer el valor de la densidad de potencia, como se ha comentado al comienzo de este apartado, así que lo almacenamos en la variable “dp0”.

El siguiente paso es ejecutar el programa en Matlab que se encargará de calcular la temperatura máxima, la temperatura media y el gradiente.

Implementación Simulador_termico.m

```
function [BT, T, rmax, rmean, rgrad]=simulador_termico(floorplan_file)
    xDoc = xmlread(floorplan_file);
    xRoot = xDoc.getDocumentElement;
    cell_size = round(str2double(xRoot.getAttribute('CellSize')));
    max_length = round(str2double(xRoot.getAttribute('Length')));
    max_width = round(str2double(xRoot.getAttribute('Width')));
    maxHeight = round(str2double(xRoot.getAttribute('NumLayers')));
    xmlBlockList = xDoc.getElementsByTagName('Block');
    Floorplan = [];

   for k = 0:xmlBlockList.getLength-1
   xmlBlock = xmlBlockList.item(k);
   id = xmlBlock.getAttribute('id');
   name = xmlBlock.getAttribute('name');
   x = xmlBlock.getAttribute('x');
   y = xmlBlock.getAttribute('y');
   z = xmlBlock.getAttribute('z');
   l = xmlBlock.getAttribute('l');
   w = xmlBlock.getAttribute('w');
   % p = xmlBlock.getAttribute('p');
   dp = xmlBlock.getAttribute('dp0');
   p = str2double(dp)*str2double(l)*str2double(w)*cell_size*(1.0e-
6)*cell_size*(1.0e-6);
   block =
struct('id',round(str2double(id)),'name',char(name),'layer',round(str2
double(z)+1),'x',cell_size*round(abs(str2double(x))),'y',cell_size*rou
nd(abs(str2double(y))),'width',cell_size*round(str2double(l)),'height'
,cell_size*round(str2double(w)),'power',p);
   floorplan = [floorplan;block];
end
[BT,T] =
simulador(floorplan,cell_size*max_length,cell_size*max_width,maxHeight
);
size(T)
for z = 0:maxHeight-1
    idxBase = 1 + 4*z;
    tempPerLayer =
T(max_length*max_width*idxBase+1:1:max_length*max_width*(idxBase+1));
```

```
   for y = 0:max_width-1
        for x = 0:max_length-1
             value="%d"/>',x,y,z,tempPerLayer(max_width*y+x+1)));
        end
   end
end

[rmax,rmean,rgrad] =
print_results(max_width,cell_size,max_length,cell_size,floorplan,T);
end
```

Este programa llamado *simulador_termico* toma el archivo XML de entrada, lee cada una de sus filas (que corresponden con un registro del Banco de registros) y calcula para cada uno de ellos los resultados correspondientes.

En la ilustración Figura 27 se ve un ejemplo de ejecución del programa *simulador_termico* para el archivo *game_life.xml* mostrado anteriormente.

```
>> [tmax, tmean, tgrad] = simulador_termico('Ejemplos/game_life_8x4.xml');
Elapsed time is 0.001000 seconds.

error_norm =

    0.0419


error_norm =

    0.0012


error_norm =

  3.4040e-005


error_norm =

  6.8580e-007

Elapsed time is 1.453000 seconds.

ans =

        1152           1
```

Figura 27 Ejemplo de ejecución

Como se desean conocer los valores de temperatura máximos, medios y el gradiente de temperatura, los obtendremos con los comandos predefinidos de Matlab.

Para la temperatura máxima, se hallará el máximo de todos los máximos (*max*) obtenidos en la ejecución y almacenados en *tmax.* Para la temperatura media se calculará la mediana (*mean*) de todos los valores medios almacenados en la variable *tmean*. Por último, el valor del gradiente es único y se devuelve en la variable definida como *tgrad*, en la Figura 28 se ilustran estos pasos.

```
>> max(tmax)

ans =

  342.0471

>> mean(tmean)

ans =

  308.0186

>> tgrad

tgrad =

  347.2783
```

Figura 28 Obtención de los resultados

Y por último, en la Figura 29 se muestra el Banco de registros resultante para el archivo de entrada.

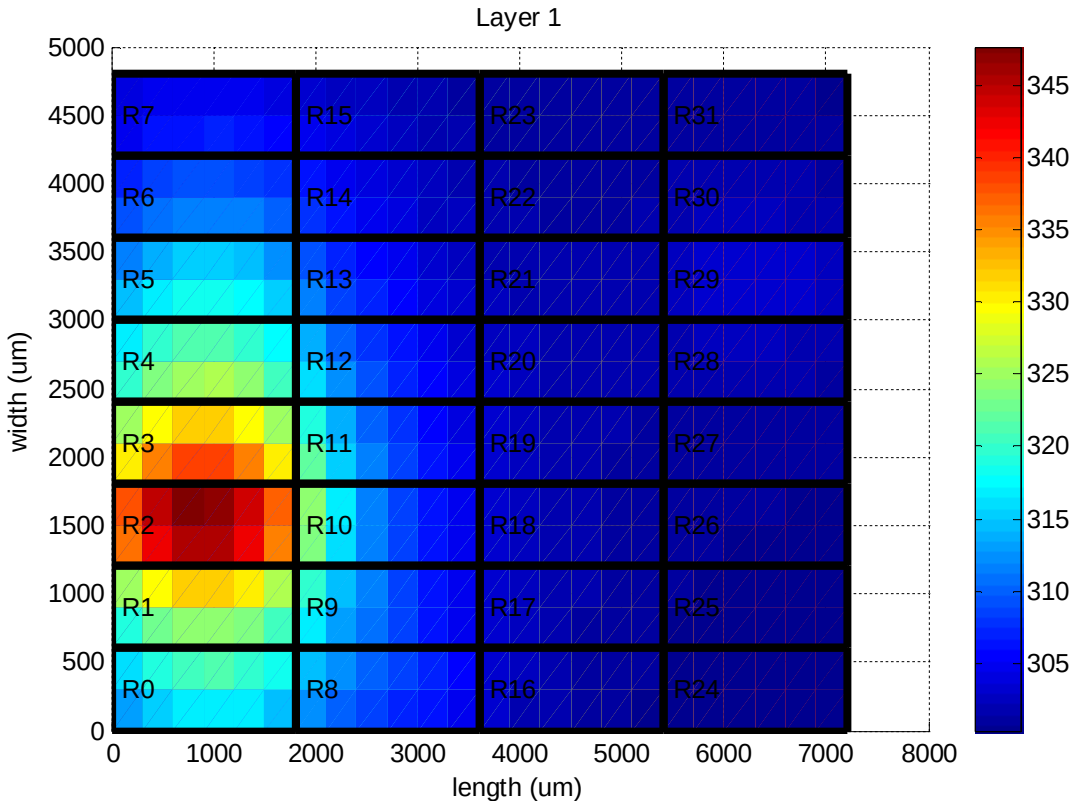

Figura 29 Banco de Registros resultante

Podemos ver que en el eje X se representa la longitud de cada una de las celdas, y por ende la del Banco de registros. En el eje Y se mide la anchura de cada uno de los registros, y a la derecha observamos una columna con un código de colores. Entendemos que cuando la temperatura es baja el registro se colorea de azul oscuro, éste color se va aclarando a medida que sube la temperatura hasta llegar a un verde claro, que representa un valor medio de la temperatura. Por último, en la parte superior de la columna se representan los registros cálidos coloreados de rojo.

En nuestro ejemplo el registro que más calor desprende, alrededor de 340 K, y se puede ver que los registros que se encuentran a su lado se ven afectados por su elevada temperatura.

En un primer vistazo, se puede suponer que si los registros R1, R2 y R3 estuvieran separados entre sí, el calor desprendido por el Banco de registros sería bastante menor.

En el apartado de *Resultados* se estudiarán con más detalle los distintos ejemplos utilizados para realizar el estudio térmico, objeto de este proyecto.

8. Resultados

En este apartado se van a mostrar detalladamente la ejecución de tres ejemplos que hemos utilizado para nuestro estudio, y que se describen a continuación.

1. El primer ejemplo se trata del algoritmo de **Ordenación de la Burbuja (o Bubblesort)**, muy conocido en los cursos de programación.

   Este algoritmo funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.

   Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo el más sencillo de implementar.

2. **El juego de la vida** es un autómata celular diseñado por el matemático John Horton Conway en el año 1970, y que simula de cierta manera el mundo en el que vivimos.

   Es un juego de cero jugadores, lo que quiere decir que su evolución está determinada por el estado inicial y no necesita ninguna entrada de datos posterior.

   El tablero es una malla formada por cuadrículas (células), cada una de estas tiene 8 células vecinas que son las que están próximas a ellas, incluso las diagonales. Las células tienen 2 estados: vivas o muertas, el estado de la malla evoluciona de manera discreta.

   El estado de las células viene determinado por sus células vecinas mediante las siguientes reglas:

   - Una célula muerta, si tiene exactamente 3 células vecinas vivas, nace el siguiente turno.
   - Una célula viva, con 2 o 3 células vecinas vivas, sigue viva. Si sólo tiene un vecino muere por soledad, mientras que si tiene más de 3 células vecinas muere por superpoblación.

   Desde un punto de vista teórico, es interesante porque es equivalente a una máquina universal de Turing, es decir, todo lo que se puede computar algorítmicamente se puede computar en el juego de la vida.

   Desde su publicación, ha atraído mucho interés debido a la gran variabilidad de la evolución de los patrones. Se considera que la vida es un buen ejemplo de emergencia y auto organización. Es interesante para los científicos,

matemáticos, economistas y otros observar cómo patrones complejos pueden provenir de la implementación de reglas muy sencillas.

3. Para el último ejemplo se ha implementado el **producto escalar** de dos vectores. En este caso concreto se trata de 2 vectores de enteros de 4000 posiciones, ya que no utilizamos la operación *producto* se aplicarán sucesivas sumas para calcular el resultado.

En casa caso se analizan tres configuraciones físicas de los registros:

1. 32x1 → 32 registros en 1 sola columna.
2. 16x2 → 2 columnas de 16 registros cada una.
3. 8x4 → 4 columnas de 8 registros cada una.

Con esto pretendemos demostrar que, aunque la disposición física de los registros varíe, nuestra metodología es fácilmente aplicable.

Además, para cada algoritmo y disposición física de registros, optimizamos la reasignación de los mismos utilizando nuestro algoritmo de optimización. Finalmente, probamos que con la metodología propuesta obtenemos un descenso en temperatura de 3.19 %.

8.1. Bubblesort

Comenzaremos con el **primer caso**. Al ejecutar en Matlab el simulador térmico, se obtienen los datos de la temperatura máxima, la mediana de las temperaturas y el valor del gradiente de temperatura.

A continuación, mostramos la ejecución:

```
[tmax,tmean,tgrad]= simulador_termico('Ejemplos/bubble_sort_32x1.xml')

Elapsed time is 0.006000 seconds.

[tmax, tmean, tgrad]= [320.7114, 305.5141, 323.0247]
```

Tras esta ejecución el programa muestra la configuración del banco de registros, y mediante un código de colores se indica la temperatura de cada uno de los registros.

Como se verá a continuación en la Figura 31 se usan siempre los primeros registros, así que se puede intuir que si estos se redistribuyeran por todo el banco de registros la temperatura máxima se vería reducida.

Ahora queremos aplicar la optimización térmica a este ejemplo. Para ello hemos creado un directorio llamado Genetic Algorithm que contiene las utilidades básicas para la optimización de la asignación de los registros.

En la consola, nos trasladamos a este directorio y ejecutamos el siguiente comando:

java -jar FloorplanGenetic?.jar bubble_sort_32x1.xml 100 250 1

Tal y como se muestra en la siguiente imagen.

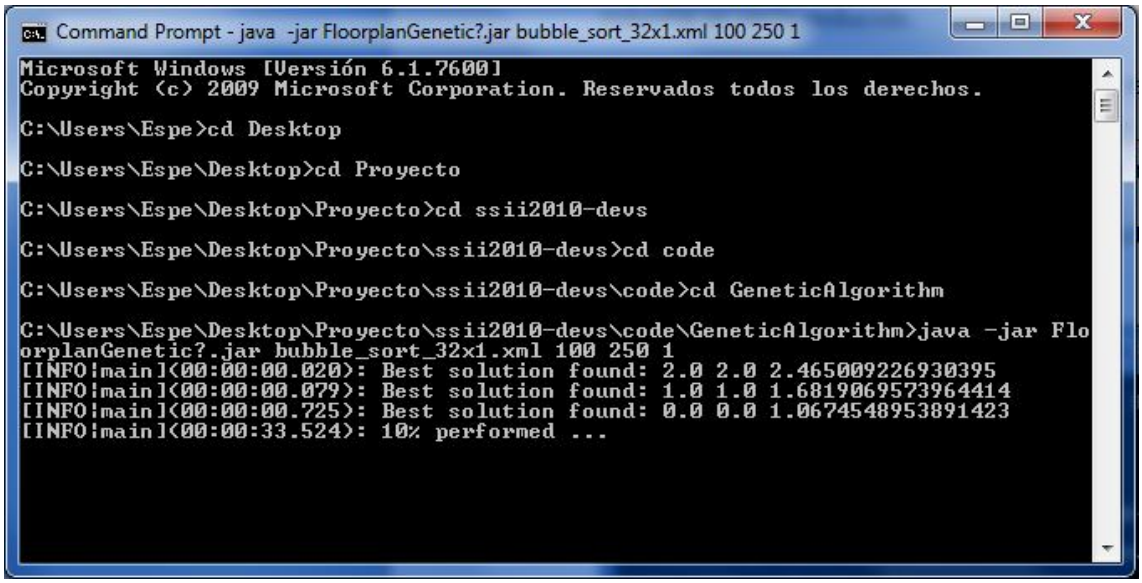

Figura 30 Estudio térmico bubble_sort 32x1

Tras un tiempo de ejecución, se crea un archivo en nuestro directorio que ejecutaremos de nuevo en el simulador térmico.

```
[tmax,tmean,tgrad]=simulador_termico ('Ejemplos/
    bubble_sort_32x1_FloorplanGenetic_0.0_0.0_0.5043783920331883.xml')
Elapsed time is 0.006395 seconds.
[tmax, tmean, tgrad]= [313.6823, 305.6045, 314.8381]
```

En la Figura 32 se muestra la distribución optimizada de los registros en el banco de registros, para la configuración 32x1 del algoritmo *bubble_sort*.

Los registros más cálidos se han distribuido por todo el banco de registros, así que cuando el compilador antes accedía al registro R0 ahora accederá al registro R14. Así con todos los demás.

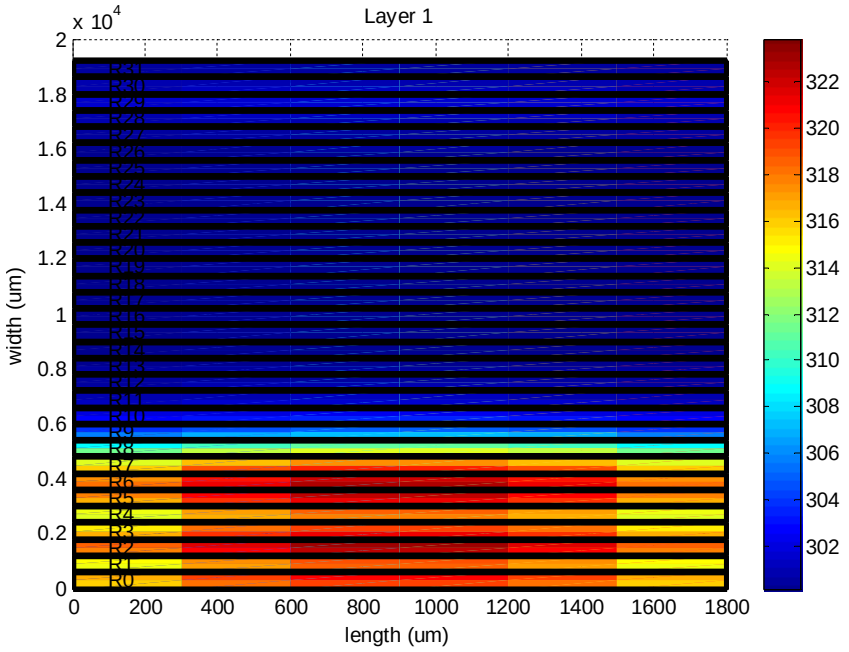

Figura 31 Bubble_sort 32x1

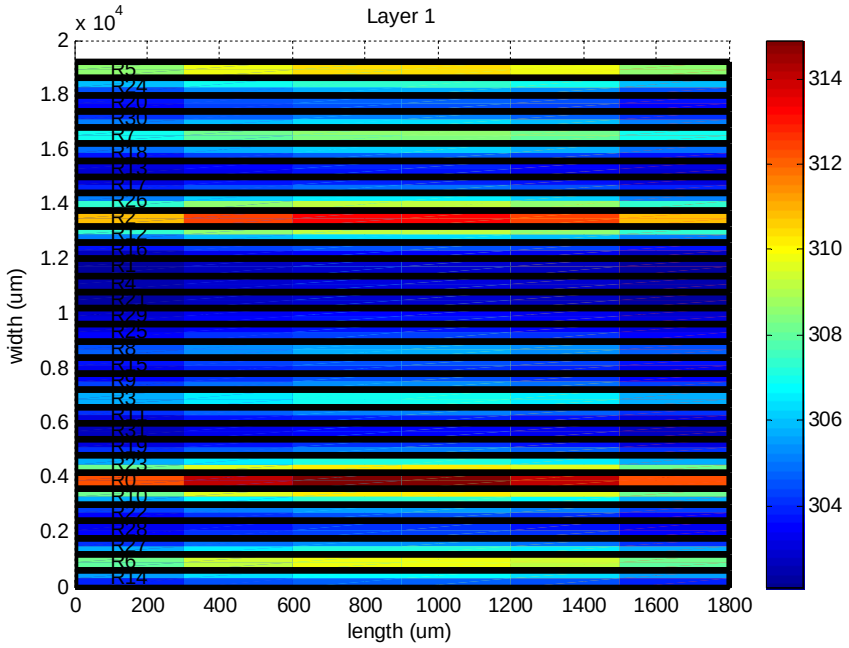

Figura 32 Bubble sort 32x1 optimizado

Para el **segundo caso**, es decir para la distribución 16x2, el proceso que seguimos es el mismo. Los resultados se muestran a continuación.

```
[tmax,tmean,tgrad]= simulador_termico('Ejemplos/bubble_sort_16x2.xml')

Elapsed time is 0.001000 seconds.

[tmax, tmean, tgrad]= [320.7542, 307.4562, 323.1206]
```

En la imagen Figura 33 se muestra la configuración resultante de ejecutar el código anterior.

Para el estudio térmico, realizamos la operación en la consola tal y como se hizo para la configuración anterior, salvo que se cambia el archivo de entrada.

java -jar FloorplanGenetic?.jar bubble_sort_16x2.xml 100 250 1

Se crea un archivo que servirá como parámetro de entrada al *simulador_termico* de Matlab.

```
[tmax,tmean,tgrad]=simulador_termico('Ejemplos/
    bubble_sort_16x2_FloorplanGenetic_0.0_0.0_0.7378006662402243.xml')

Elapsed time is 0.006248 seconds.

[tmax,tmean,tgrad]= [314.9498, 360.5345, 316.2305]
```

Como se ilustra en la Figura 34, el banco de registros ha reorganizado los registros de manera que los componentes más cálidos se encuentran separados entre sí.

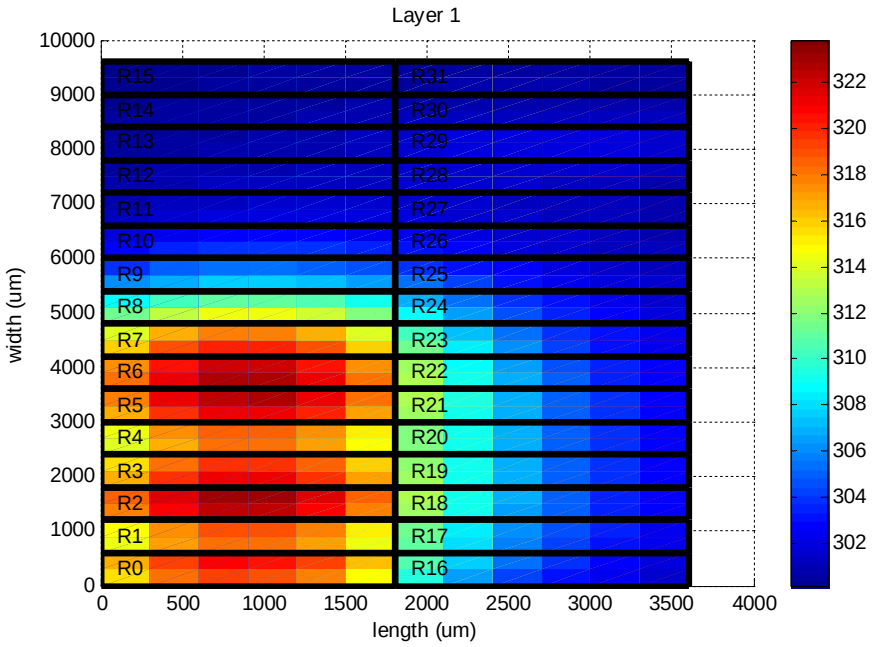

Figura 33 Bubble sort 16x2

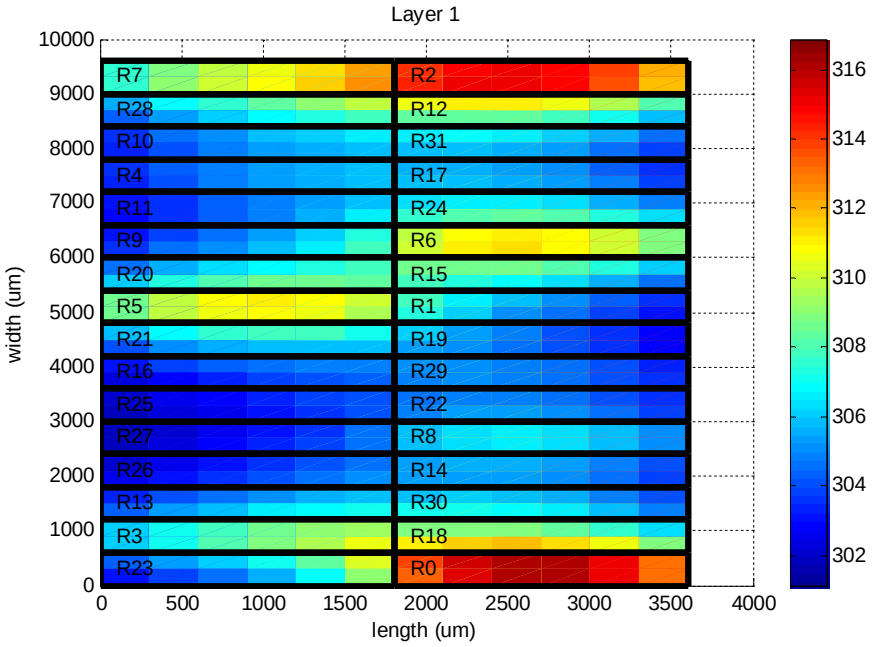

Figura 34 Bubble sort 16x2 optimizado

Por último en el **tercer caso** la distribución es de 8x4, salvo por eso la ejecución es exactamente igual que en los dos casos anteriores.

Primero ejecutamos en Matlab el *simulador_termico* para obtener la temperatura máxima, la mediana y el gradiente de temperatura. El resultado de esta ejecución se ve reflejado en la Figura 35.

```
[tmax,tmean,tgrad]= simulador_termico ('Ejemplos/bubble_sort_8x4.xml')

Elapsed time is 0.001000 seconds.

[tmax,tmean,tgrad]= [320.7542, 307.4562, 323.1206]
```

Como podemos ver, en las tres configuraciones de este ejemplo, la primera configuración del banco de registros es prácticamente igual y que siempre se usan los 8 primeros registros, ya que se trata del mismo programa.

Se ejecuta el algoritmo genético para la redistribución de los registros.

java -jar FloorplanGenetic?.jar bubble_sort_8x4.xml 100 250 1

Ejecutamos ahora el archivo solución en el *simulador_termico* y obtenemos los valores de las temperaturas nuevos y la organización del banco de registros.

```
[tmax,tmean,tgrad]=simulador_termico('Ejemplos/
    bubble_sort_8x4_FloorplanGenetic_0.0_0.0_0.7390326237167513.xml')

Elapsed time is 0.000771 seconds.

[tmax,tmean,tgrad]= [314.4087, 306.4656, 315.6918]
```

La nueva configuración del banco de registros se muestra en la Figura 36.

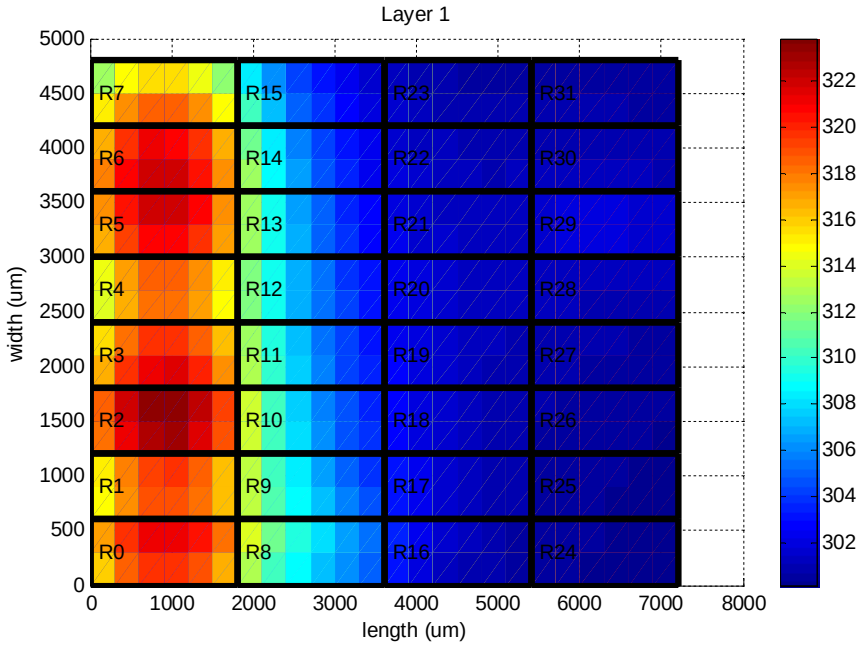

Figura 35 Bubble sort 8x4

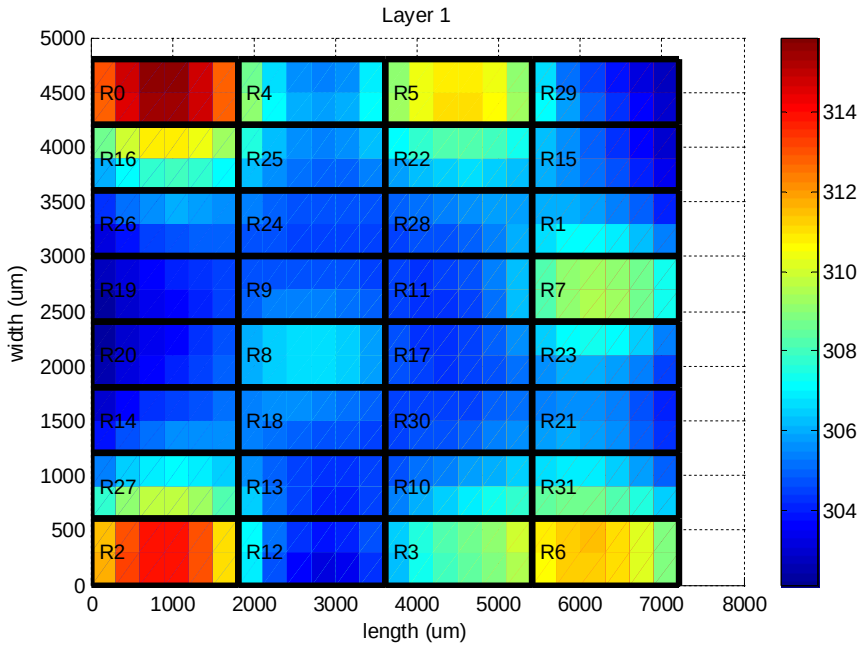

Figura 36 Bubble sort 8x4 optimizado

Por último, obtenemos la siguiente tabla resumen con los valores de las temperaturas obtenidos en los pasos anteriores.

Tabla 1 Bubble sort 32x1 Resumen

Configuración	Tmax	Tmean	Tgrad
Original	320.7114 K	305.5141 K	323.0247 K
Optimizado	313.6823 K	305.6045 K	314.8381 K

Tabla 2 Bubble sort 16x2 Resumen

Configuración	Tmax	Tmean	Tgrad
Original	320.7542 K	307.4562 K	323.1206 K
Optimizado	314.9498 K	306.6345 K	316.2305 K

Tabla 3 Bubble sort 8x4 Resumen

Configuración	Tmax	Tmean	Tgrad
Original	321.1659 K	306.9405 K	323.4912 K
Optimizado	314.4087 K	306.4656 K	315.6918 K

Respecto a la temperatura máxima, el mayor descenso se observa en la distribución 32x1, cuantificado en un 2.2 %. Bajo esta distribución el compilador debería por ejemplo reasignar el registro 14 al registro 1

8.2. Game of life

Para el **primer caso** partimos de un archivo XML similar al explicado en el apartado 5 y ejecutamos en Matlab el *simulador_termico*. A continuación se muestra el estado del banco de registros inicial y los valores de las temperaturas.

```
[tmax,tmean,tgrad]= simulador_termico ('Ejemplos/game_life_32x1.xml')

Elapsed time is 0.006000 seconds.

[tmax,tmean,tgrad]= [342.0913, 305.9509, 347.1630]
```

En la Figura 38 se puede ver cómo los registros más utilizados por el compilador están muy cerca entre sí, por lo que además del calor desprendido por si temperatura también se ven influenciados por sus vecinos.

Ejecutamos el simulador térmico, encargado de la reasignación de registros, mediante la consola y tomando como partida el archivo de entrada de *simulador_termico*.

java -jar FloorplanGenetic?.jar game_life_32x1.xml 100 250 1

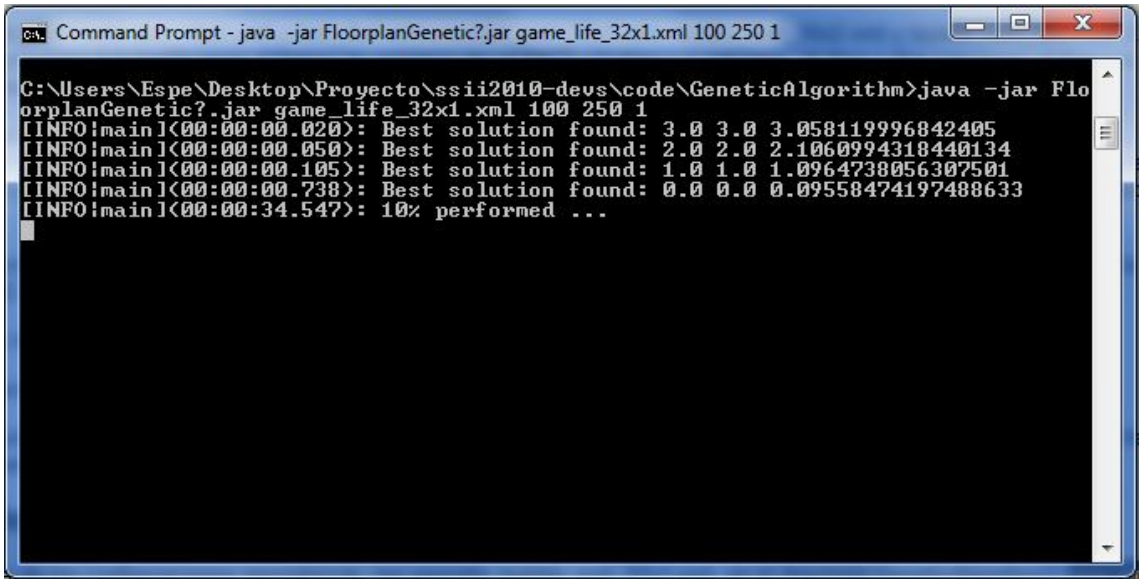

Figura 37 Estudio térmico Game life 32x1

El fichero resultante lo tomaremos ahora como entrada al *simulador_termico*, el cual nos volverá a dar como resultado el valor de las temperaturas y la nueva organización del banco de registros.

```
[tmax,tmean,tgrad]=simulador_termico ('Ejemplos/
    game_life_32x1_FloorplanGenetic_0.0_0.0_0.061046041142991704.xml')

Elapsed time is 0.006344 seconds.

[tmax,tmean,tgrad]= [335.2312, 305.3040, 338.5822]
```

El nuevo banco de registros se puede observar en la Figura 39.

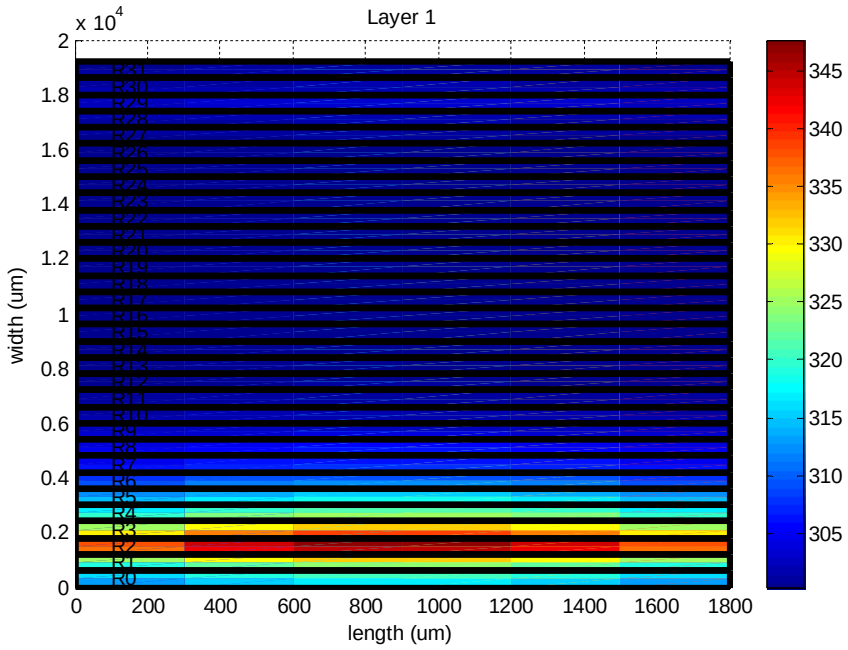

Figura 38 Game life 32x1

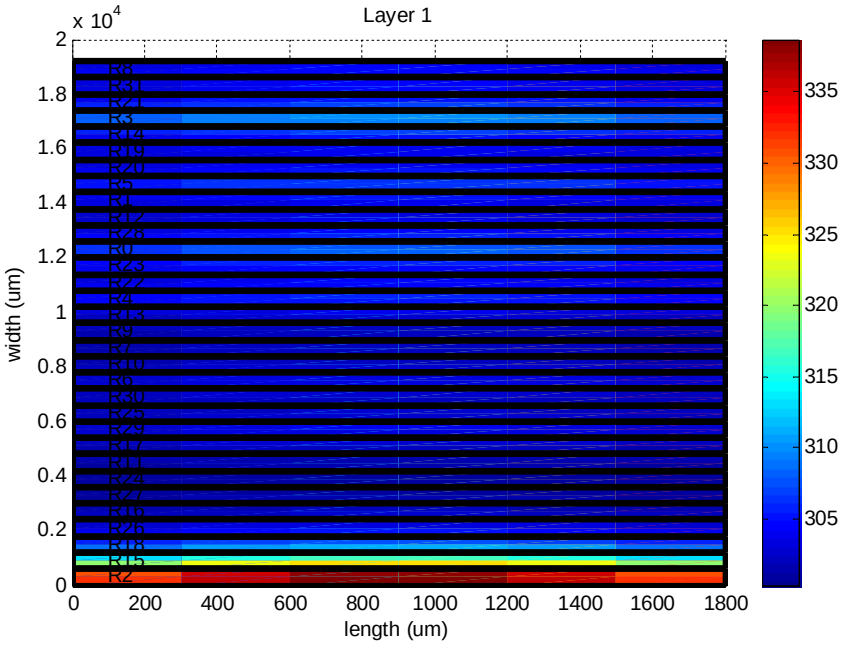

Figura 39 Game of life 32x1 optimizado

El **segundo caso** representa la configuración 16x2, 16 filas de registros distribuidas en 2 columnas. Primero ejecutamos el *simulador_termico* con el archivo XML inicial para obtener los valores de las temperaturas y la configuración del banco de registros.

La nueva organización del banco de registros viene representada en la Figura 40.

```
[tmax,tmean,tgrad]= simulador_termico ('Ejemplos/game_life_16x2xml')

Elapsed time is 0.000000 seconds.

[tmax,tmean,tgrad]= [341.6040, 308.0080, 346.9836]
```

Ejecutamos el algoritmo genético para nuestro estudio térmico, el proceso por consola es similar que en el caso 1 así que directamente se muestran los valores de las temperaturas y la nueva organización del banco de registros.

java -jar FloorplanGenetic?.jar game_life_16x2.xml 100 250 1

```
[tmax,tmean,tgrad]=simulador_termico ('Ejemplos/
    game_life_16x2_FloorplanGenetic_0.0_0.0_0.09539995622479185.xml')

Elapsed time is 0.000724 seconds.

[tmax,tmean,tgrad]= [334.2306, 306.6166, 347.2783]
```

Como muestra la Figura 41, los registros más problemáticos, es decir R1, R2 y R3 han sido redistribuidos por todo el banco de registros, por lo tanto obtenemos la configuración mostrada en la figura anterior.

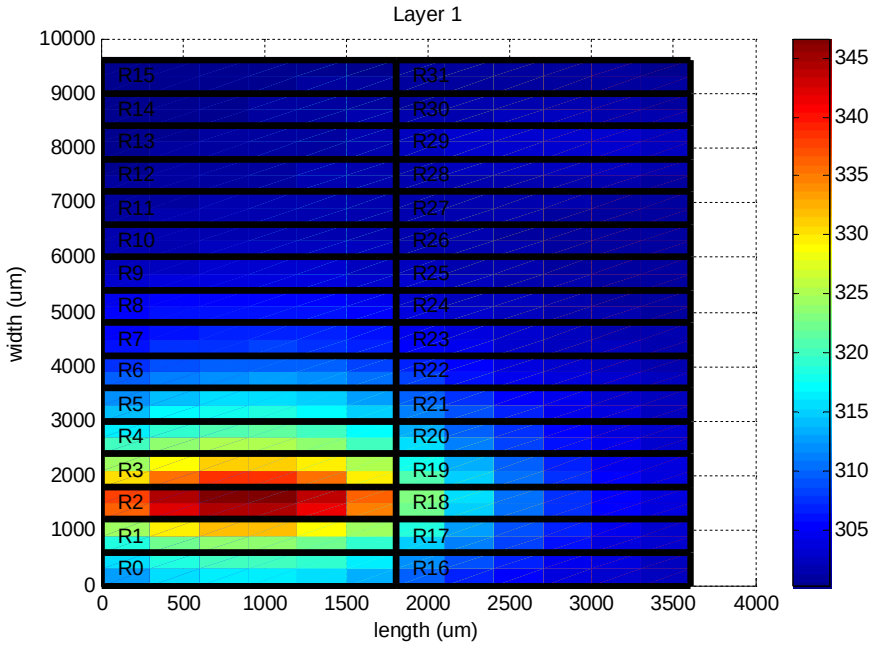

Figura 40 Game life 16x2

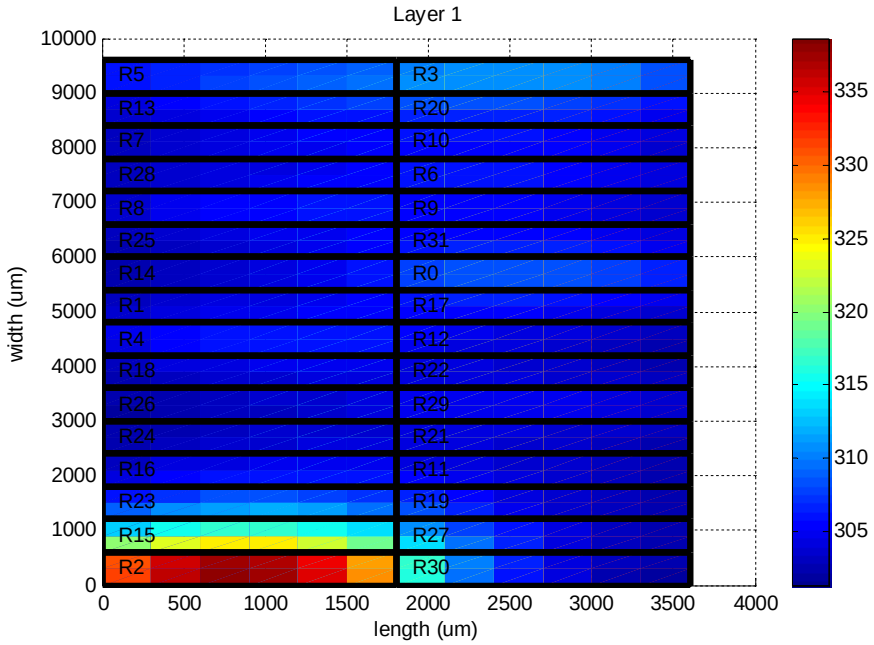

Figura 41 Game life 16x2 optimizado

El **tercer caso** representa la configuración 8x4 del banco de registros. Comenzamos el proceso ejecutando el *simulador_termico* en Matlab tomando el archivo XML como parámetro de entrada.

Al ejecutar el *simulador_térmico* en Matlab, obtenemos los siguientes resultados de la temperatura máxima, media y el valor del gradiente.

```
[tmax,tmean,tgrad]= simulador_termico ('Ejemplos/game_life_8x4xml')

Elapsed time is 0.000516 seconds.

[tmax,tmean,tgrad]= [342.0471, 308.0186, 347.2783]
```

Como en los casos anteriores, aplicamos el algoritmo genético a nuestro fichero de entrada y se creará otro fichero con la configuración optimizada del banco de registros.

java -jar FloorplanGenetic?.jar game_life_8x4.xml 100 250 1

```
[tmax,tmean,tgrad]=simulador_termico ('Ejemplos/
    game_life_8x4_FloorplanGenetic_0.0_0.0_0.10220990694143361.xml')

Elapsed time is 0.000546 seconds.

[tmax,tmean,tgrad]= [334.2575, 306.4543, 338.1704]
```

Se toma este fichero resultado como la nueva entrada del *simulador_termico*, las siguientes imágenes Figura 42 y Figura 43 muestran las dos organizaciones del banco de registros obtenidas.

Como en los anteriores casos, la temperatura máxima se ha visto reducida gracias a la nueva organización del banco de registros.

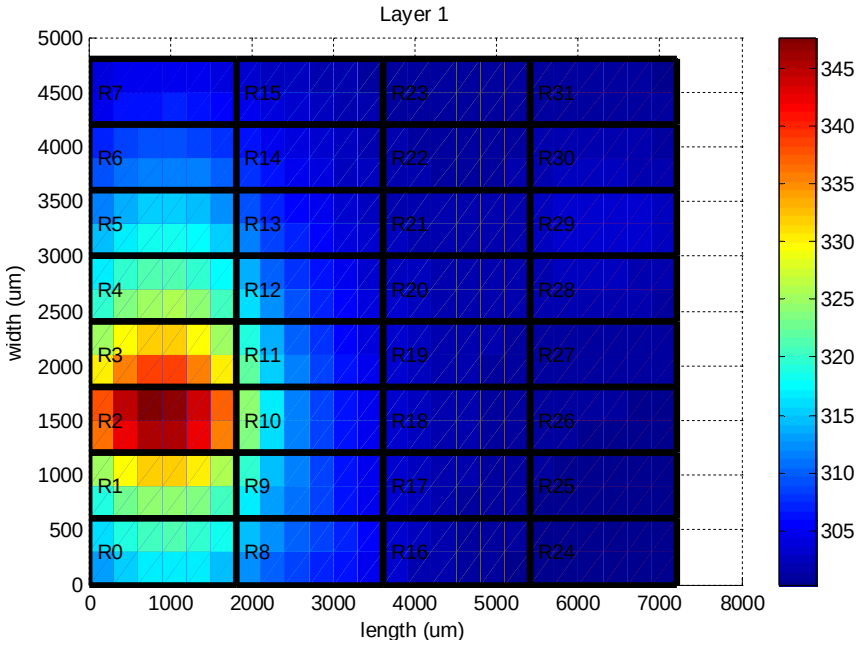

Figura 42 Game life 8x4

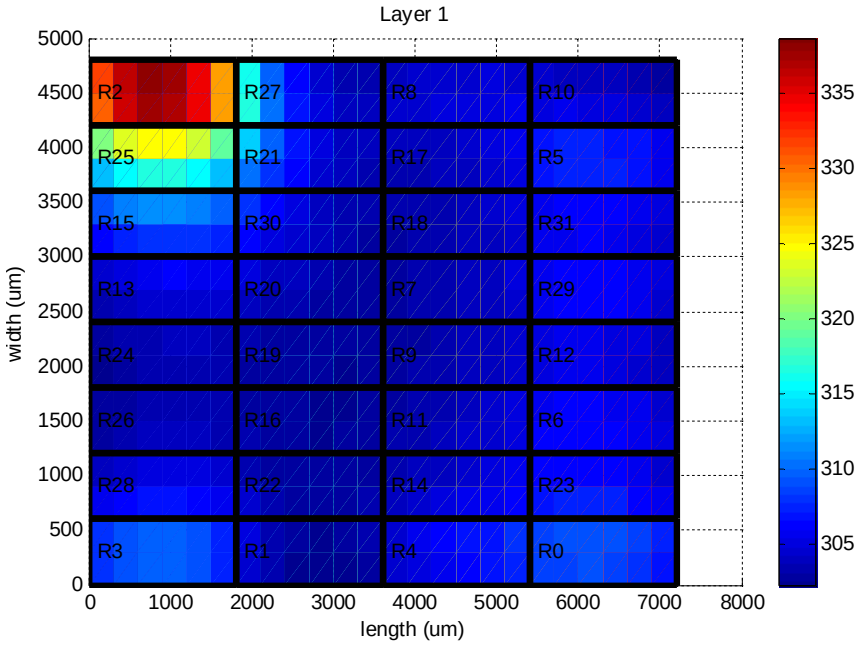

Figura 43 Game life 8x4 optimizado

Por último, obtenemos la siguiente tabla resumen con los valores de las temperaturas obtenidos en los pasos anteriores.

Tabla 4 Game life 32x1 Resumen

Configuración	Tmax	Tmean	Tgrad
Original	342.0913 K	305.9509 K	347.1630 K
Optimizado	335.2312 K	305.3040 K	338.5822 K

Tabla 5 Game life 16x2 Resumen

Configuración	Tmax	Tmean	Tgrad
Original	341.6040 K	308.0080 K	346.9836 K
Optimizado	334.2306 K	306.6166 K	338.1590 K

Tabla 6 Game life 8x4 Resumen

Configuración	Tmax	Tmean	Tgrad
Original	342.0471 K	308.0186 K	347.2783 K
Optimizado	334.2575 K	306.4543 K	338.1704 K

Respecto a la temperatura máxima, el mayor descenso se observa en la distribución 8x4, cuantificado en un 2.28 %. Bajo esta distribución el compilador debería por ejemplo reasignar el registro 26 al registro 2.

8.3. Prodesc

En el **caso 1**, al estar todos los registros distribuidos a lo largo de una columna, éstos sólo se verán afectados por los vecinos superior e inferior. Al ejecutar el *simulador_termico* obtenemos los valores de las temperaturas y la configuración original del banco de registros.

```
[tmax,tmean,tgrad]= simulador_termico ('Ejemplos/prodesc_o1_32x1.xml')

Elapsed time is 0.000000 seconds.

[tmax,tmean,tgrad]= [332.1638, 305.9103, 336.7758]
```

En la Figura 44 los registros R2, R3, R4 y R5 son los más utilizados y por tanto los más cálidos de todo el banco de registros. Al aplicar el algoritmo genético para realizar el estudio térmico, se observará en la Figura 45 que estos 4 registros serán distribuidos.

java -jar FloorplanGenetic?.jar prodesc_o1_32x1.xml 100 250 1

```
[tmax,tmean,tgrad]= simulador_termico ('Ejemplos/
prodesc_o1_32x1_FloorplanGenetic_0.0_0.0_0.14664567032463657.xml')

Elapsed time is 0.000894 seconds.

[tmax,tmean,tgrad]= [321.9847, 305.1894, 3240.0249]
```

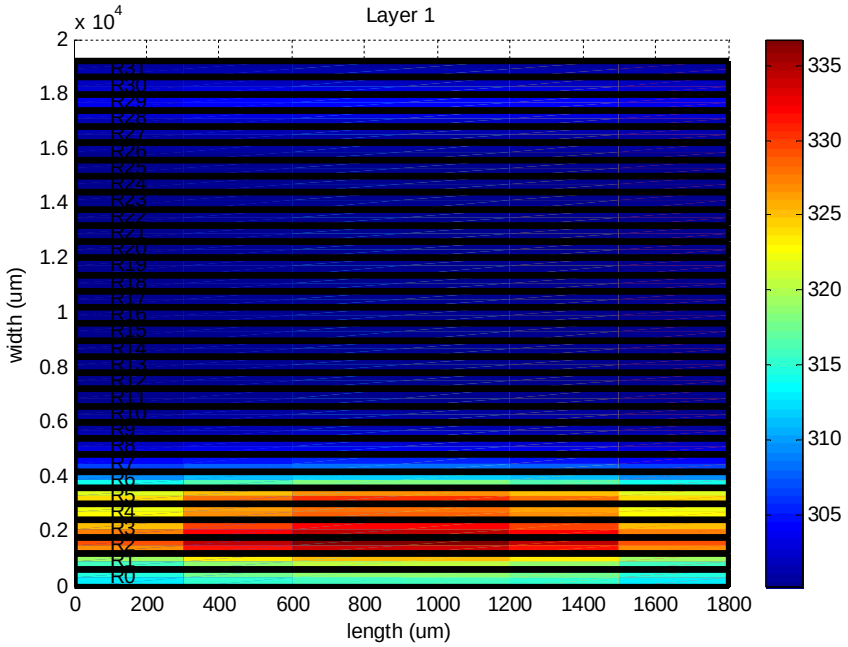

Figura 44 Producto escalar 32x1

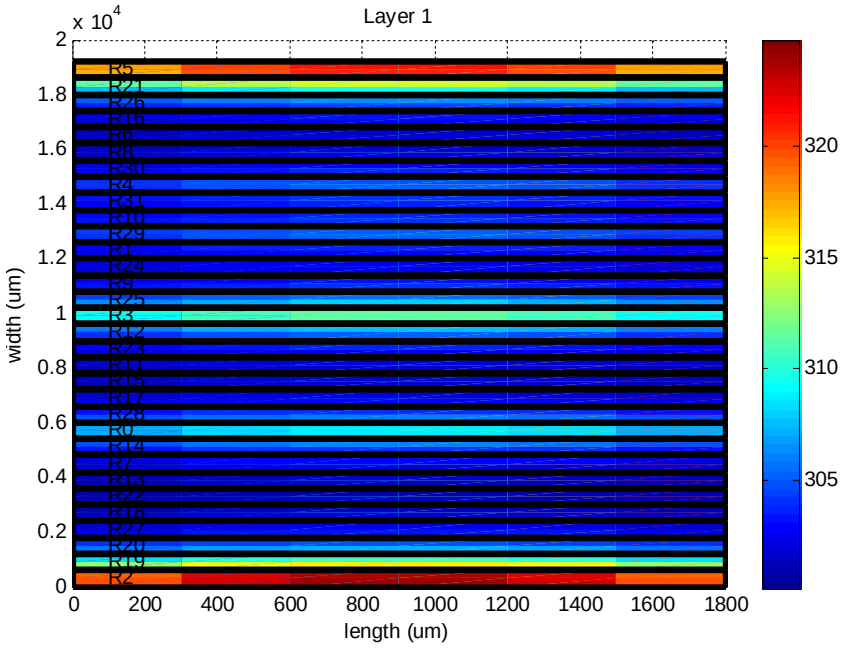

Figura 45 Producto escalar 32x1 optimizado

En la Figura 45 se muestra lo que se predijo observando la configuración del banco de registros anterior, tanto R2, R3, R4 como R5 han sido redistribuidos por todo el banco. Observamos que en realidad los registros más cálidos eran R2 y R5, así que la temperatura tan alta de R3 y R4 era como consecuencia de sus registros vecinos.

Para el **caso 2** se parte de la configuración 16x2, así que los registros se verán afectados además de por sus vecinos superior e inferior, por la temperatura de un registro a su lado.

Como se ha hecho en los anteriores ejemplos, se ejecuta el *simulador_termico* en Matlab tomando como entrada el fichero XML que representa esta configuración.

```
[tmax,tmean,tgrad]= simulador_termico ('Ejemplos/prodesc_o1_16x2.xml')

Elapsed time is 0.000000 seconds.

[tmax,tmean,tgrad]= [332.0173, 307.9749, 336.7343]
```

Ahora en la Figura 46 se ve que los registros R18, R19, R20 y R21 también ven su temperatura aumentada por culpa de sus registros vecinos.

Al ejecutar el algoritmo genético observamos un cambio en la distribución de los registros.

java -jar FloorplanGenetic?.jar prodesc_o1_16x2.xml 100 250 1

```
[tmax,tmean,tgrad]= simulador_termico ('Ejemplos/

prodesc_o1_16x2_FloorplanGenetic_0.0_0.0_0.2592652554275275.xml')

Elapsed time is 0.000890 seconds.

[tmax,tmean,tgrad]= [321.6156, 306.7106, 323.9549]
```

La temperatura se ha reducido de 332.0173 K a 321.6156 K, los registros más cálidos en la configuración no optimizada han sido redistribuidos por todo el banco de registros. Esto se observa en la Figura 47.

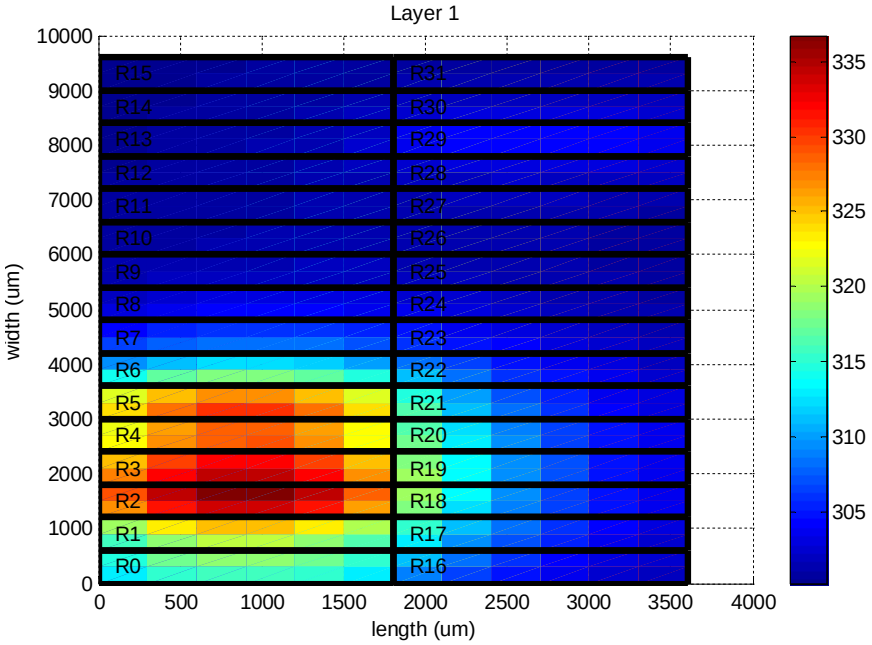

Figura 46 Producto escalar 16x2

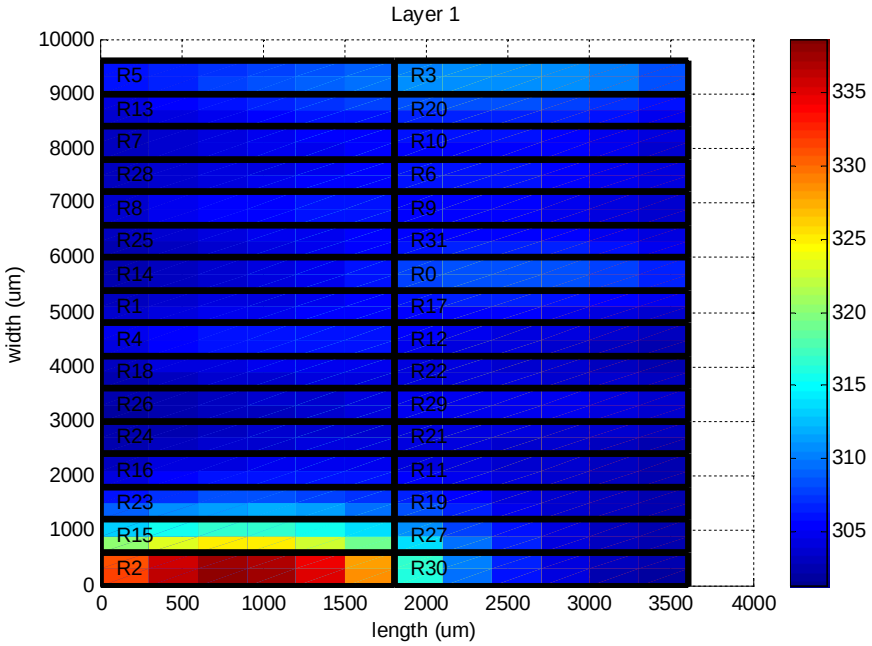

Figura 47 Producto escalar 16x2 optimizado

Por último tenemos el **caso 3**, que representa la configuración 8x4. Como tenemos 4 columnas de registros, ahora estos se verán afectados con respecto a su temperatura por vecinos de los cuatro puntos cardinales.

Al ejecutar el *simulador_termico* con esta configuración como parámetro de entrada, obtenemos los siguientes resultados.

```
[tmax,tmean,tgrad]= simulador_termico ('Ejemplos/prodesc_o1_8x4.xml')

Elapsed time is 0.000000 seconds.

[tmax,tmean,tgrad]= [332.0289, 307.9826, 336.7450]
```

Los registros afectados colateralmente en este caso son R10, R11, R12 y R13. Hay que observar que ahora los registros R18, R19, R20 y R21 no ven modificada su temperatura debido a que la distancia a los registros cálidos es mayor. Esta conclusión se obtiene observando la Figura 48.

Para realizar la redistribución del banco de registros se aplica el algoritmo genético, como en casos anteriores, y se observan los resultados.

java -jar FloorplanGenetic?.jar prodesc_o1_8x4.xml 100 250 1

```
[tmax,tmean,tgrad]= simulador_termico ('Ejemplos/

prodesc_o1_8x4_FloorplanGenetic_0.0_0.0_0.2546737737025825.xml')

Elapsed time is 0.000000 seconds.

[tmax,tmean,tgrad]= [321.4236, 306.2690, 323.8060]
```

Como se ilustra en la Figura 49, la temperatura del banco de registros optimizada ha pasado de 332.0289 K a 321.4236 K, es la consecuencia directa de redistribuir los registros más críticos.

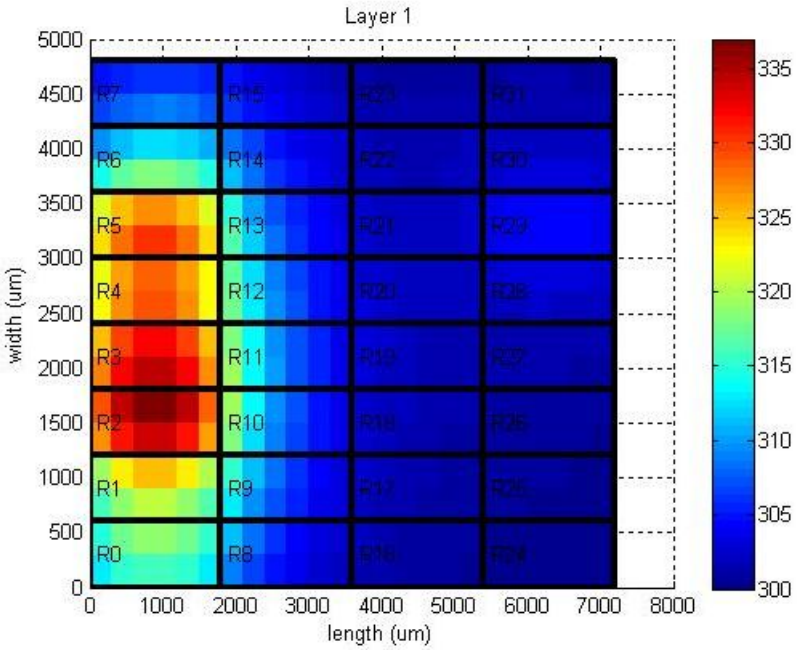

Figura 48 Producto escalar 8x4

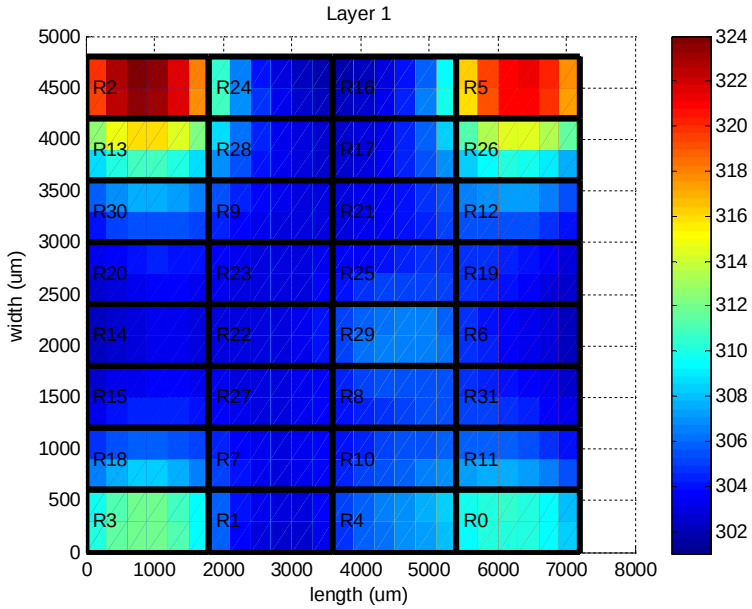

Figura 49 Producto escalar 8x4 optimizado

Por último, obtenemos la siguiente tabla resumen con los valores de las temperaturas obtenidos en los pasos anteriores.

Tabla 7 Producto escalar 32x1 Resumen

Configuración	Tmax	Tmean	Tgrad
Original	332.1683 K	305.9103 K	336.7758 K
Optimizado	321.9847 K	305.1894 K	324.0249 K

Tabla 8 Producto escalar 16x2 Resumen

Configuración	Tmax	Tmean	Tgrad
Original	332.0173 K	307.9749 K	336.7343 K
Optimizado	321.6156 K	306.7106 K	323.9549 K

Tabla 9 Producto escalar 8x4 Resumen

Configuración	Tmax	Tmean	Tgrad
Original	332.0289 K	307.9826 K	336.7450 K
Optimizado	321.4236 K	306.2690 K	323.8060 K

Respecto a la temperatura máxima, el mayor descenso se observa en la distribución 8x4, cuantificado en un 3.19 %. Bajo esta distribución el compilador debería por ejemplo reasignar el registro 30 al registro 5.

Conclusiones

Las actuales escalas de integración introducen nuevos fenómenos que degradan considerablemente la fiabilidad de los chips. Electromigración, consumo de potencia, rendimiento y temperatura son algunos de los parámetros a tener muy en cuenta hoy en día. En este trabajo hemos abordado el problema térmico, y más concretamente su impacto en el banco de registros de un procesador. Para reducir este impacto, hemos desarrollado una metodología que, mediante políticas de reasignación de registros, disminuyen considerablemente la temperatura final del mismo.

Esta metodología consiste en primer lugar, en un análisis de uso de los registros de las aplicaciones originales compiladas con un compilador estándar (gcc). Con este análisis podemos determinar el consumo de potencia de cada uno de los registros y, por ende, el impacto térmico de la aplicación. Finalmente, mediante una optimización, obtenemos una política de reasignación de registros que tiene como objetivo principal distribuir uniformemente la temperatura, rediciendo así puntos calientes.

Aunque la metodología propuesta es aplicable a cualquier arquitectura destino, los ejemplos propuestos en este trabajo se han ejecutado sobre un simulador del procesador MIPS32. El simulador lo hemos construido con técnicas conocidas de simulación de eventos discretos, en concreto DEVS, que permite un modelado software orientado a componentes de cualquier sistema. El simulador permite obtener el perfil de consumo de potencia mencionado anteriormente. Más tarde, hemos desarrollado un modelo térmico de varias configuraciones del banco de registros del MIPS32. El análisis de las distintas configuraciones permite constatar que la metodología propuesta es aplicable a cualquier disposición física del banco de registros, lo que de nuevo valida la independencia del proceso seguido con respecto a la arquitectura destino.

El algoritmo de reasignación de registros implementado está basado en computación evolutiva. Este algoritmo utiliza el modelo térmico para evaluar la bondad de las diferentes políticas de reasignación, que evolucionan de forma natural mediante operadores clásicos de los sistemas bioinspirados como selección, cruce o mutación.

El análisis de los resultados permite constatar que una buena política de reasignación puede reducir la temperatura del banco de registros hasta en torno a 10 °K, lo que representa un 3,2 % de la temperatura máxima en la política de asignación original.

Glosario

Algoritmo genético: Conjunto de instrucciones que hacen evolucionar una población de individuos sometiéndola a acciones aleatorias semejantes a las que actúan en la evolución biológica (mutaciones y recombinaciones genéticas), así como también a una selección de acuerdo con algún criterio, en función del cual se decide cuáles son los individuos más adaptados, que sobreviven, y cuáles los menos aptos, que son descartados.

DEVS: Es un acrónimo del inglés para referirse a Discrete Event System Specificatio (Especificación de Sistemas de Eventos Discretos). Es un término estándar en el campo de la Simulación para modular y analizar sistemas de diversos tipos, en particular, sistemas de eventos discretos y sistemas híbridos continuos y discretos.

Estudio térmico: Es la medida de los cambios físicos o químicos que ocurren en una sustancia en función de la temperatura mientras la muestra se calienta con un programa de temperaturas controlado.

MIPS: Son las siglas de Microproccesor without Interlocked Pipeline Stages y es una familia de microprocesadores de arquitectura RISC (Reduced Instructions Set Computer) desarrollados por MIPS Technologies y usados en muchos sistemas integrados.

MOEA: Son las siglas de Multi-Objetive Evolutionary Algorithm, que representa el conjunto de algoritmos evolutivos multiobjetivos.

Referencias

- Arnaldo Lucas I., “Power profiling-guided floorplanner for thermal optimization in 3d multiprocessor architectures”, Universidad Complutense de Madrid, 2011.

- Brooks D., Dick R.P., Joseph R., Shang L., "Power, thermal, and reliability modeling in nanometer-scale microprocessors", IEEE Computer Society, 2007.

- Calvo Valdés F., Roldán Ramírez J., San Miguel Sánchez A., “Simulador del procesador MIPS sobre el formalismo DEVS”, Universidad Complutense de Madrid, 2010.

- Deb K., Pratap A., Agarwal S., Meyarivan T.A., “Fast and Elitist Multiobjetive Genetic Algorithm: NSGA-II”, IEEE Transactions on Evolutionary Computation, 6 182-197, 2002.

- Patterson, D.A. and J.L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 3ª edition, Morgan Kaufmann, 2007.

- Qadri M.Y., McDonald-Maier K. D., “Data cache-energy and throughput models: Design exploration for embedded processors”, EURASIP Journal on Embedded Systems, 2009.

- Risco-Martín J.L, “Multi-objective genetic algorithm”, 2011.

- Yeager K.C., "The Mips R10000 superscalar microprocessor," Micro, IEEE , vol.16, no.2, pp.28-41, 1996